RCA 1800
MICROSYSTEMS

BASIC3
High-Level-Language Interpreter
CDP18S841 User Manual

MPM-841A Suggested Price $5.00

BASIC3
High-Level-Language Interpreter
CDP18S841 User Manual

RCA Solid State

Brussels • Buenos Aires • Hamburg • Madrid • Mexico City • Milan
Montreal • Paris • Sao Paulo • Somerville NJ • Stockholm
Sunbury-on-Thames • Taipei • Tokyo

The software described in this publication is copyrighted by the RCA Corporation.

Information furnished by RCA is believed to be accurate and reliable. However, no responsibility is assumed by RCA for its use; nor for any infringements of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights or RCA.

Copyright 1982 by RCA Corporation
(All rights reserved under Pan-American Copyright Convention)

Trademark(s)®Registered
Marca(s) Registrada(s)

Printed in USA/2-82

Foreword

The BASIC3 Interpreter CDP18S841 is supplied in two versions. The first version is designed for use on an RCA COSMAC Microboard Computer Development System (MCDS) equipped with a Cassette Tape Drive Mechanism. The second version, a subset of the first, called Run-Time BASIC CDP18S842 is designed for use in custom-made systems comprised of RCA Microboard modules.

This Manual describes the BASIC3 language and gives detailed operation information for the interpreter. The Manual assumes that the reader is familiar with the CDP1802 Microprocessor and the COSMAC Development System. Users requiring additional information on the available hardware and software support systems should refer to the COSMAC technical literature. Reference should also be made to the technical data sheets for the specific devices. The following publications are of particular interest.

User Manual for the CDP1802 COSMAC Microprocessor, MPM-201

User Manual for the RCA COSMAC Microboard Computer Development System (MCDS) CDP18S693 and CDP18S694, MPM-293

Technical Data Sheet for the CDP1802A COSMAC Microprocessor, File No. 1305

Users requiring additional information or an instruction guide on the Basic language should refer to one of the many books available on the subject. Suggested references include:

James S. Coan, **BASIC BASIC; An Introduction to Computer Programming in Basic Language,** Hayden Books, 1978

Robert L. Albrecht et al., **BASIC,** John Wiley and Sons, 1978

Richard Forsyth, **The BASIC Idea,** Chapman and Hall, 1978

Seymour Hirsh, **BASIC; A Programmed Text,** John Wiley and Sons, 1975

Contents

Page

1. Elements of BASIC 3 7
   - Memory Requirements 7
   - Running BASIC 3 7
   - Programs, Statements, and Functions 8
   - Numbers 9
   - Operators 11
     - Arithmetic Operators 11
     - Assignment Operators 11
     - Relational Operators 11
     - Logical Operators 11
     - Miscellaneous Operators 11
   - Strings and Arrays 12
2. Statements 13
   - Command Statements 13
     - BYE 13
     - CLD 13
     - CLS 13
     - EDIT 13
     - ENINT, DISINT 14
     - EOD 14
     - EOP 14
     - FORMAT 15
     - LIST 15
     - NEW 15
     - RENUMBER 15
     - RUN, RUN expr, RUN+ 15
     - TRACE 15
   - Comment and Definition Statements 16
     - DEFINT 16
     - DEFUS 16
     - DEG 16
     - DIM 16
     - FIXED 17
     - RAD 17
     - REM 17
   - Assignment Statements 17
     - DMAPT 17
     - LET 17
     - POKE 17
     - STQ 17
   - Control Statements 17
     - END 17
     - EXIT 17
     - FOR 18
     - GOSUB 18
     - GOTO 18
     - IF 19

Page

     - NEXT 19
     - RETURN 19
     - WAIT 19
   - Program Data Statements 19
     - DATA 19
     - READ 20
     - RESTORE 20
   - I/O Statements 20
     - INPUT 20
     - OUT 20
     - PRINT or PR 21
   - Saving or Loading a Program or Data on Disk Statements 21
     - PLOAD 21
     - PSAVE 21
     - DSAVE 21
     - DLOAD 21
   - Sequential Device I/O Statements 21
     - RFLN, WFLN 22
     - DIN, DOUT 22
     - TIN, TOUT 22
     - CLOSE 22
   - Machine Language Subroutine Statement 22
     - CALL 22
3. Functions 23
   - Arithmetic Functions 23
     - ABS 23
     - ATN 23
     - COS 23
     - EXP 23
     - FNUM 23
     - INT 23
     - INUM 23
     - LOG 23
     - MOD 23
     - PI 24
     - RND 24
     - SGN 24
     - SIN 24
     - SQR 24
   - String Functions 24
     - ASC 24
     - CHR$ 24
     - FVAL 24
     - LEN 24
     - MID$ 25

Contents (cont'd)

Page

I/O Functions 25
EF 25
INP 25
MEM 25
PEEK 25
QST 25
TAB 25
Machine Language Function 26
USR 26
4. Creating and Editing Programs 27
5. Error Messages and Program Debugging 29
6. Run-time BASIC CDP18S842 30
Using Run-time BASIC 30
Memory Requirements 31
Summary of Run-time BASIC Usage 32

Page

Appendix A—Error Messages and Their Meanings 33
Appendix B—Backus-Naur Format (BNF) ... 34
Basic Definitions 35
BNF Syntax of BASIC 3 35
Appendix C—Work Page Allocation Details . 38
Appendix D—ASCII-HEX Table 39
Appendix E—Summary of Statements 40
Appendix F—Summary of Functions 41
Appendix G—Alphabetical Index of BASIC 3 Statements and Functions 42
Appendix H—Using the Cassette Tapes 43
Appendix I—Program for Converting from Pseudocode into ASCII 44

1. Elements of BASIC3

This chapter provides an introduction to the BASIC3 language and describes its parameters. Although BASIC3 is a high-level language that provides the statements and functions required by the experienced programmer, it is an interactive language and has the significant virtue that it can be easily learned and readily used by the beginning programmer. Supplied only as an interpreter, the speed of BASIC3 approaches that of a compiler by translating the statements as they are entered into a pseudocode. It is this pseudocode that is actually executed.

Memory Requirements

The BASIC3 interpreter is stored in ROM in the memory space from B000H to DFFFH (hexadecimal notation). The interpreter requires additional RAM in low memory beginning at 0000H. The amount of RAM available in low memory controls the size of the programs that may be written. The locations 000H through 020FH are used as work space by the interpreter. When the system is first initialized, the interpreter begins a search of memory from 7FFFH downward for RAM and establishes a stack at the top of the last page. The user program and program-generated data are located between the work space and the stack. The map of memory is shown in Fig. 1.

The first page of the work space is used by the interpreter for line buffers and constant storage; the second page is for variable storage and computations. See Appendix C for detailed work page allocations. The suffixed variables, array variables, and strings are stored in RAM immediately after the end of the program. This area is dynamically allocated as program line numbers are added, deleted, or edited.

Running BASIC3

BASIC3 is entered by means of the monitor program. When the user types "B", BASIC3 responds with the message

```
C RCA 81
BASIC3 - V0.0
C/W?
```

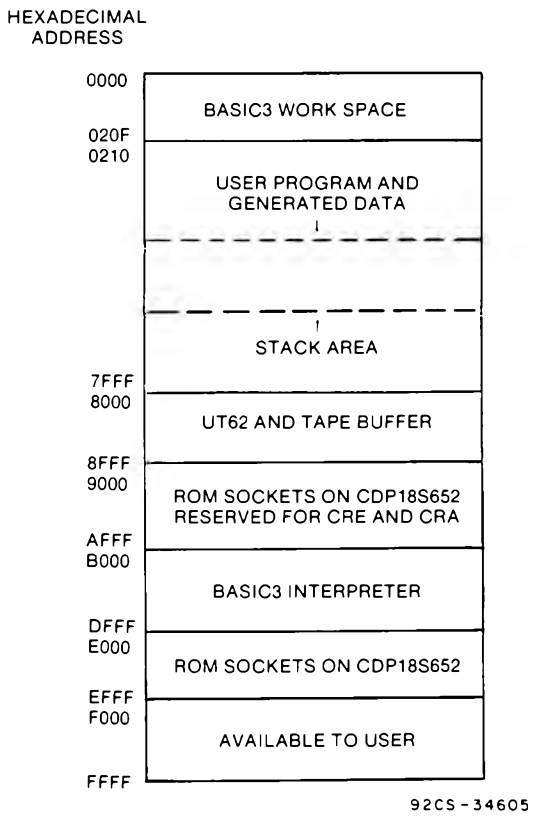


Fig. 1 - Memory allocations.

BASIC3 now waits for the user to type "C" for cold start or "W" for warm start. A cold start clears all data and program areas. A warm start allows the user to exit and re-enter BASIC3 without destroying a program. Care must be taken, however, not to modify lower memory if the user wishes to re-enter by way of the warm start. Any of the monitor commands may be used without directly overwriting any previously entered user program.

After the user types "C", the system responds

```
CREADY (Warm start response is WREADY)
:
```

The colon is the BASIC3 prompt and indicates that BASIC3 is ready to accept input.

Most BASIC3 statements may be executed in one of two ways, direct execution or program execution. When a program statement is typed in without a line number, the statement is executed immediately. For example, typing in the following

```
: PRINT "THIS IS A TEST"
```

results in the following immediate response

```
THIS IS A TEST
READY
:
```

The statements that may be directly executed are listed in Table I.

The program mode of execution is entered when the statement is preceded by a line number. Execution of these statements does not occur until the program is run by the direct execution of a RUN statement.

Programs, Statements, and Functions

A number of terms should be explained before a detailed description of BASIC3 is given. A BASIC3 program is a collection of BASIC3 **statements.** Table I lists the executable statements in all of their forms. It also shows whether each is directly executable, program executable, or both. The following is a sample program in BASIC3:

```
10 FOR A=1 TO 3
20 PRINT "A=";A
30 NEXT A
40 END
```

Each statement of the program is preceded by a line number. The line numbers may be any integer in the range of 1 to 65534. The use of a number greater than 65534 results in program destruction. Each line may

Table I—Statements

MNEMONICS	DIR. EXEC.	PROG. EXEC.	MNEMONICS	DIR. EXEC.	PROG. EXEC.
BYE	X	X	INPUT		X
CALL	X	X	LET	X	X
CLD	X	X	LIST	X	X
CLOSE	X	X	NEW	X	
CLS	X	X			
DATA		X	NEXT		X
DEFINT	X	X	OUT	X	X
DEFUS	X		PLOAD	X	
DEG	X	X	POKE	X	X
DIM	X	X	PR	X	X
DIN	X	X	PRINT	X	X
DISTINT	X	X	PSAVE	X	X
DLOAD	X	X	RAD	X	X
DMAPT	X	X	READ		X
DOUT	X	X	REM	X	X
DSAVE	X	X	RENUMBER	X	X
EDIT	X		RESTORE		X
END	X	X	RETURN		X
ENINT	X	X	RFLN	X	X
EOD	X	X	RUN	X	
EOP	X	X	RUN+	X	
EXIT		X	STQ	X	X
FIXED	X	X	TIN	X	X
FOR		X	TOUT	X	X
FORMAT	X	X	TRACE		X
GOSUB		X	WAIT	X	X
GOTO		X	WFLN	X	X
IF	X	X			

consist of one or more statements with each statement separated by a colon. A line, however, cannot exceed 95 characters.

The above program could have been written as follows:

```
10 FOR A=1 TO 3:PRINT "A=";A:NEXT A
20 END
```

In the above programs, the END statement is used but it is not required. In BASIC3, an END statement can be used to terminate execution at any point within a program but is not required as the final statement.

A statement represents an instruction to the interpreter and frequently requires an **operand.** In the statement

```
20 PRINT A ,
```

PRINT is the instruction and A is the operand. The operand can be a function, expression, variable, string, string variable, or string expression. These operands are defined as follows:

Functions are key words that return a value to BASIC3. They may stand alone or be used with other functions to form expressions. A list of the functions is given in Table II. Some of the functions are restricted as to how or where they may be used. TAB, for example, may only be used in a PRINT statement. Any restrictions are pointed out in the individual function descriptions given in a subsequent chapter.

The term **expression** indicates any arithmetic expression. It may consist of anything from a single number, a single variable, or a function, to a group of functions tied together with arithmetic operators (also described subsequently). All expressions evaluate to a single number value. The following are some examples of expressions:

```
5
A
K(5)
4*K(5)
SIN(45)
3*SIN(45) ↑ .5
3*(2+A)/(B+C)
```

The mode of an expression is defined by the mode of the first term in that expression, as discussed under Mixed-Mode Operation.

The term **variable** refers to any of the 6682 variables A to Z, and variables A_0-Z_0 through A_{255}-Z_{255}, or the 26 array names A(expr) to Z(expr). The simple variables A-Z are allocated storage on work page #2 (See Appendix C) making them global variables. Each variable is four bytes long. They are not affected by the CLD statement and preserve their values with each invocation of the PLOAD statement. This feature is also very useful for passing parameters between BASIC3 and machine language subroutines.

The term **string** refers to any string of alphanumeric characters that are enclosed in quotation marks.

The term **string expression** refers to any string or string array, any string of characters, the string function or any concatenated strings.

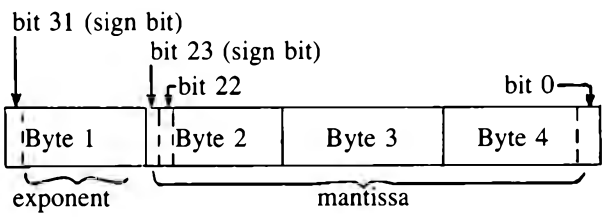
Table II—Functions*

ABS	INP	PI
ASC	INT	QST
ATN	INUM	RND$
CHR$	LEN	SGN
COS	LOG	SIN
EF	MEM	SQR
EXP	MID	TAB
FNUM	MOD	USR
FVAL	PEEK	

*All functions are executable directly and by program.

Numbers

BASIC3 is capable of handling both integer and floating-point numbers. Both types are stored as 32-bit signed numbers. In the case of floating-point numbers, eight bits define the exponent and 24 bits the mantissa. The floating-point representation in BASIC3 consists of four bytes. Bytes 2 - 4 are the mantissa.



Of the 24 bits representing the two's-complement mantissa, the most significant bit (MSB) represents the sign (0 is positive, 1 is negative) and the remaining 23 bits are a normalized 23-bit fraction. Normalization adjusts the exponent so that the "implied" most significant digit of the mantissa magnitude is a 1 or the mantissa is zero. The "implied" bit does not appear in the floating-point representation. It is shifted out to the left before the program stores the numbers in the internal representation.

The exponent consists of 1 byte. It is expressed as excess 128 (base 2). That is, the power to raise the base is equal to the exponent field minus 128. The MSB of the exponent also indicates the sign of the exponent. When it is 1, the exponent is positive. When it is 0, the exponent is negative.

Some examples of the floating-point representation and their decimal equivalent are given below:

Hexadecimal				Decimal
FF	7F	FF	EF	0.170141E+39
83	20	00	00	5.0
81	00	00	00	1.0
80	00	00	00	0.5
01	00	00	00	0.293873E−38
01	FF	FF	FF	0.587747E−38
00	00	00	00	0
80	80	00	00	−0.5
83	A0	00	00	−5.0
FF	FF	FF	FF	−0.170141E+39

The range of numbers is

Integer: −2147483647 to +2147483647

Floating point: −.170141E39 to +.170141E39

Integer numbers are accurate over the entire range, but floating-point numbers are accurate to approximately six mantissa digits, although up to nine digits are allowed on data entry.

The following are examples of valid numbers:

1	1E02
−3	−3E−10
2.5	−.4E8
−6.8	1.25E-4

The following are invalid:

E02	no mantissa
.12345678902E+5	too many digits

All numbers entered from the keyboard are stored as they are entered. If entered without a decimal point, they are stored as integer; if entered with a decimal point or with "E" notation, they are stored as floating point. If both data types occur in the same expression, i.e., mixed mode, the first term of the expression determines the mode of the remainder of the expression. If a number begins an expression, the expression is assumed to be floating point regardless of whether the number is entered in integer or floating point.

If a variable begins the expression, the mode of the variable (see DEFINT) sets the mode of the expression. Suppose that the variable "A" is set to integer. Consider the following program as an example:

```
5  DEFINT A
10 A=10
20 PR A/20
30 PR 5/A
```

Note that in line number 10, the variable "A" sets that expression as integer because "A" had been defined as integer. "A" now contains an integer 10. In line number 20, BASIC3 will output an integer 0 instead of .5 because "A" set that expression to integer. In line number 30, BASIC3 would respond with a decimal .5 because the expression began with a number. In this case, the variable "A" was recognized as integer but was converted to floating point.

Consider the following example:

```
10 DEFINT A
20 A=23.2
30 PRINT A
```

In line number 20, variable "A" sets the mode as integer. The number 23.2 is recognized as floating point. It is, therefore, converted to integer and stored in "A". It is important to note that when a floating-point number or function is converted to integer, it is rounded to its nearest whole number. Line number 30, then, would result in the printing of an integer 23 with no decimal point.

It should be noted that the mode of the variable in the FOR/NEXT loop sets the mode of the start, stop, and step. Also, the mode of an IF/THEN statement is set by the first term in the equality or inequality.

If an expression begins with a function (SIN, COS, SQR, PEEK, USR, INP, etc.), it is assumed to be floating point. If any of these expressions are encountered in an already defined integer expression, they will be interpreted as integer automatically. The only exceptions are RND(arg) and INUM(arg) which are designated integer numbers at all times. If these functions are encountered in a floating-point expression, a mixed-mode error will result. Similarly, RND and FNUM(expr) are designated floating-point functions and cause a mixed-mode error when encountered in an integer expression.

In any of the functions that involve arguments, the arguments are expressions in themselves. Their mode is independent of the mode of the expression in which the function resides. For example:

```
10 DEFINT A
20 LET A=10
30 PR 20*SQR(A*10.3)
```

Line number 20 stores an integer in variable A. In line number 30, the expression A*10.3 is evaluated as integer resulting in 100 as the argument for the SQR function. The main expression in the PR statement is defined as floating point because it begins with a number. The result, therefore, is 200. in floating point. Again, the mode of the expression contained in the argument is independent of the mode of any other expression.

In the following examples and their results, it is assumed that the variables A, B, and C have been defined as integer and that all others are floating point.

```
10 A=2
20 PR SQR(49.3)*A
```

results in:

```
14.0428
```

Next example:

```
10 A=2
20 PR A*SQR(49.3)
```

results in:

```
14
```

Next example:

```
PR A*RND
```

results in ERR CODE 7 (a mixed-mode error) because A sets the expression as integer and RND with no argument must be floating point.

```
PR RND*A
```

results in a random floating-point number between 0. and 1. multiplied by the value of A.

```
PR A*RND(10)
```

results in a random integer number in the range of 0 to 10 multiplied by A.

```
PR 2*RND(10)
```

results in a mixed-mode error. Note that mixed-mode errors are possible only with the misuse of RND, INUM, and FNUM. All other cases are taken care of by BASIC3.

If an expression begins with a hexadecimal number (#XX or @XXXX) it is designated as integer because most operations using hexadecimal numbers are integer in nature. If a hexadecimal number is encountered within a floating-point expression, however, it will be converted to floating point.

Operators

A list of the BASIC3 operators is given in Table III. These operators can be divided into five categories: arithmetic, assignment, relational, logical, and miscellaneous. A description of the operators in each of these categories follows:

Arithmetic Operators. The following list gives the arithmetic operators with their symbols and an indication of the priority of each in mathematical operations.

Unary minus	:	−	:first level priority
Exponentiate	:	↑	:first level priority
Multiply	:	*	:second level priority
Divide	:	/	:second level priority
Add	:	+	:third level priority
Subtract	:	−	:third level priority

Operators on the same level of priority are acted upon from left to right. The symbol "-" is also used to denote negative numbers. Some examples illustrating the exercise of priorities follow.

4*3 ↑ 2=36
4/2*3=6
4+2*3−5=5

Assignment Operator. The assignment symbol is the equals sign "=". When this symbol is used for the purpose of assignment, it takes on the definition "takes the value of" rather than "equals". Some examples are:

LET A=5
LET B=B+A*2

Relational Operators. The relational operators are listed below.

Equality :=
Inequality :<>
Greater Than : >
Less Than : <
Greater Than or Equal To: >=
Less Than or Equal To : <=

Expressions that contain relational operators are evaluated to true or false rather than to a numeric value. Some examples are:

IF A=3+B ↑ 2 THEN
IF A<>4 THEN
IF A*2<=B/4 THEN

Logical Operators. The logical operators are AND, OR, XOR, NOT. These operators perform either the logical operation on the given expressions or the logical complement of the given expression. Some examples are:

IF A AND B <> C THEN
FOR I=A AND B TO A OR B STEP A/B
PR A XOR B
GOTO NOT(A)

It should be noted that the logical operators convert the expressions to integer before performing the indicated operation and that the expression following the NOT operator must be within parentheses. Further, the logical operators have the same level of priority as + and − . Following are some examples of the NOT operator usage:

X	NOT (X)	X	NOT (X)
0	255	−1	0
0.5	254	−2	1
1	254	−128	127
2	253	−129	128
128	127	−150	149
129	126	−255	254
1024	255	−256	255
		−512	255

Miscellaneous Operators. The miscellaneous operators are listed below.

:	separates statements on the same line
;	PRINT delimiter
;,	PRINT delimiter
(	string delimiter
	group terms

Table III—Operators and Their Meaning

Symbol	Meaning
↑	exponentiation (4 ↑ 3=64) [See footnote 4]
*	multiplication
/	division
+	addition, string concatenation
−	subtraction
=	equal to, takes the value of
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
AND	Boolean operator
OR	Boolean operator
XOR	Boolean operator
NOT	Boolean operator
:	separates statements on the same line
;	PRINT delimiter
;,	PRINT delimiter (horizontal tab)
"	enclose a string of characters
(	group terms
)	group terms
. . . .	8-bit binary values (\10010110\)[1]
#	8-bit hexadecimal values[2]
@	16-bit hexadecimal values[3]

[1]Must be 8 binary digits; e.g. The binary equivalent of 2 is \00000010\.

[2]Must be 2 hexadecimal digits; e.g. The hexadecimal number A is #0A.

[3]Must be 4 hexadecimal digits; e.g. The hexadecimal number A is @000A.

[4]Negative numbers raised to positive integer powers are acceptable; e.g. − 2 ↑ 3=−8

)	group terms
\. . . .\	8-bit binary values
#	8-bit hexadecimal values
@	16-bit hexadecimal values

Some examples are:

```
PRINT A,B; "YES" : GOTO 10
LET A = (3 + 4)/2
LET B = # F8: C=@01F3
IF A <> \10011010\ THEN GOTO 100
```

Strings and Arrays

BASIC3 is equipped with 26 arrays [A(expr) to Z(expr)] and 26 strings or string arrays [A$ to Z$ and A$(expr) to Z$(expr)]. The 26 arrays may be one- or two-dimensional arrays. The maximum size of the array is 255 in any dimension. The minimum size of any dimension is 1. The largest array that can exist is G(255,255) - subject to available RAM - and the smallest is G(1). The maximum number of array elements is limited by the amount of RAM that is available. Any array that is used must first be dimensioned by use of a DIM statement, which will be explained in greater detail in the statement descriptions (Chapter 2).

The array space is cleared in several ways: RUN, NEW. and CLD. This step is discussed later in the explanation of each of these statements. It is also valuable to know how much memory is being used by each array. Each element in the array is four bytes long. Furthermore, each array has a header of five bytes. Therefore, an array that is dimensioned DIM A(10,10) contains 100 elements (10 × 10), which is 400 bytes for the array named "A". Similarly, array B, dimensioned as DIM B(20), would contain 20 elements, 80 bytes plus the header and bring the total to 85 bytes. The EOD statement, as explained later, tells how to find out how full the data space is getting.

String capability is also provided by BASIC3. It provides 26 string variables (A$ to Z$). Each string can contain up to 127 alphanumeric characters. Also included are string arrays that enable the programmer to index into a table of strings by means of some numerical argument. A string, therefore, may be referenced to by A$(N) where N is any expression (or number) with a maximum value of 255. If a number greater than 255 is used, erroneous results could occur.

A string having no argument is assumed to have an argument of zero. For example,

A$ is equivalent to A$(0).

It is also important to note that no dimensioning of any kind for strings is necessary. Memory is automatically allocated each time a string is generated. Also note that the generation of a string C$(10) does not imply that C$(1) through C$(9) exist. Only C$(10) exists and an attempt to read C$(1) results in an error message stating the C$(1) has not yet been generated.

Functions and procedures for string manipulation are also provided. A MID$ function is included to extract some portion of a previously defined string to be used in a conditional statement or in defining a new string. See the statement description for MID$. Also provided is the capability of concatenating strings, that is, to add two strings together to form a larger string. Several examples follows:

```
10 A$="NEVER"
20 B$="THE"
30 LET A$(5)="LESS"
40 LET B$(1)=A$+B$+A$(5)
50 PRINT B$(1)
```

Line number 40 would result in the creation of a new string B$(1) that contains the first three strings placed back to back. "NEVERTHELESS" would be printed at line number 50.

```
10 INPUT B$
20 A$(1)="CONTINUE"
30 A$(2)="PHASE 1"
40 A$(3)="PHASE 2"
50 IF B$=A$(1)+A$(2) GOTO 100
60 IF B$=A$(1)+A$(3) GOTO 200
70 GOTO 10
```

The above program would wait at line number 10 for a string to be entered. That string would be stored in B$. It would then be compared to A$(1)+A$(2) and A$(1)+A$(3) with program execution transferred to the appropriate spot depending upon a match or mismatch. Any string expression other than concatenation cannot involve two different string names. The commands are described in detail later.

The only limitation on string concatenation is that any string generated as the result of adding two or more previously defined strings may not exceed the 127-character limit mentioned earlier.

It has been pointed out that BASIC3 provides the capability to build large strings out of a library of smaller ones (concatenation) and to extract small strings out of larger ones (MID$ command). Also available are the following string functions:

```
CHR$(expr)
ASC(string name)
LEN(string name)
```

These functions are explained in detail later.

2. Statements

In this chapter, the BASIC3 executable statements are described. These statements are divided into eight groups and are discussed in alphabetical order under each group head. The eight groups are:

Command statements
Comment and definition statements
Assignment statements
Control statements
Program data statements
I/O statements
Tape statements
Machine language subroutine statements

Command Statements

Command statements are system directives and are normally executed in the immediate mode. The command statements include BYE, CLD, CLS, DISINT, EDIT, ENINT, EOD, EOP, FORMAT, LIST, NEW, RENUMBER, RUN, RUN+, and TRACE.

BYE

This statement transfers control from BASIC3 back to MCDS. To re-enter BASIC3, the user should type "B", as described earlier.

CLD

This statement (clear data) when executed resets all string and array pointers to their initial states. In effect, it erases all strings and arrays.

CLS

This statement (clear screen) outputs a Control D (ASCII #04) to the output routines. CLS clears the screen and homes the cursor on video terminals that respond to CONTROL D, such as the RCA COSMAC Video Terminal CDP18S040. For other terminals, the control character can be changed by a POKE statement to work page address 00ADH.

EDIT expr

This statement opens the line number called by expr and allows the user to modify the line on a character basis. On entry, the line is listed and the cursor (or print head) is returned to the beginning of the next line. The user then moves the cursor by means of the space key to a position under the printed or displayed line that is one character before the one that is to be modified. At this point three EDIT options are available. The user may type I (Insert), D (Delete), or C (Change). After the option selected is typed, the option character is displayed and the cursor moves under the character to be modified.

The **Insert** option allows characters to be entered ahead of any character position in the line. The **Delete** option removes characters from the line sequentially with each space key entry. The **Change** option replaces sequentially each character with the new one that is typed. If a (CR) is typed after an I or a C, a delete to the end of the line is performed.

After the user performs the desired modification, typing CONTROL S (DC3) causes the line to be retyped with the modification present. The user can reposition the cursor and remodify the line or exit the EDIT mode by typing CONTROL S again. Note that in one pass only one option (I, D, or C) is allowed. The user can, however, exercise each option any number of times before leaving the EDIT mode on a selected line.

CONTROL A (SOH) performs the abort function. If it is entered after the option character but before CONTROL S, the line is retyped. If it is entered before the option character, the EDIT session is aborted.

Some examples follow. In these examples, "u" denotes the space key.

```
:EDIT 10
10 PRINT "HELLO"
         ISAYu        (Space over to first "quote",
                      type I, then type
                      "SAY(space)".
                      Control S retypes the line.)
```

```
10 PRINT "SAY HELLO"
         Duuuu          (Space to "quote", delete
                        4 characters. Control S re-
                        types the line.)

10 PRINT "HELLO"
         CJERRY         (Space to "quote", Change 5
                        characters. Retype.)

10 PRINT "JERRY"
                        (Type control S to exit edit
                        session on this line.)
READY
:
```

ENINT expr
DISINT

These statements enable BASIC3 to recognize and process interrupts totally in the BASIC3 language. These statements can be used in numerous applications ranging from a powerful debugging tool to a real-time data processor. The interrupts are handled in a simple way. When BASIC3 is entered, all interrupts are automatically disabled; that is, all generated interrupts are ignored. At any time, however, either in the direct mode or in the program mode, when an ENINT expr statement is executed, all interrupts will be recognized. The expression following ENINT defines a line number in the BASIC3 program. Whenever an interrupt occurs during program execution, the current line number, which may contain many statements, is completed by BASIC3. Control is then transferred to the BASIC3 subroutine residing at the line number defined in the last ENINT statement. This "interrupt" is then executed until a RETURN statement is encountered at which time BASIC3 transfers control back to the original program where execution was interrupted. The user could reenable interrupts within the interrupt routine itself (that is, change the line number defining the interrupt routine) thereby, in effect, permitting multilevel interrupts.

When a DISINT statement is executed, the interrupts will be ignored. Once the interrupts are enabled, nothing (including a NEW) will disable the interrupts except a DISINT.

Fig. 2 is an example of implementing interrupts in BASIC3. If the BASIC3 program is interrupted, execution transfers to line #1000. A message is printed and the user can look at any of the variables or numerical array elements by means of the INPUT statement in line #1040. Responding to the INPUT statement in line #1060 with a NO transfers the program to line #1080 where the option is given to turn the TRACE mode on or off. In this way, when a long program is being either run or debugged, the user can stop it at any time, interrogate any of the variables or array elements, turn the trace on or off, and resume execution.

EOD
EOP

The EOD statement (end of data) prints the hexadecimal address of the end of data (arrays and strings). The EOP statement (end of program) is similar to EOD in that, when executed (normally in the direct execution mode), it automatically prints the hexadecimal address of the end of the current program space. The user can type it

```
  5 DIM X(1)
 10 ENINT 1000
 20       :
          :
     Body of BASIC3 program
          :
          :
1000 PR"PROGRAM INTERRUPTED!!"
1010 INPUT"SHOULD VARIABLES BE CHECKED"Z$
1020 IF MID$(Z$,1,1)="Y" GOTO 1040
1030 GOTO 1080
1040 INPUT"WHAT VARIABLE"X(1)
1050 PR X(1)
1060 INPUT"MORE?"Z$
1070 IF MID$(Z$,1,1)="Y" GOTO 1040
1080 INPUT"SHOULD TRACE BE TURNED ON"Z$
1090 IF MID$(Z$,1,1)="Y" TRACE 1: GOTO 1110
1100 TRACE 0
1110 PR"PROGRAM EXECUTION RESUMED!!"
1120 RETURN
```

Fig. 2 - An example illustrating the implementing of interrupts.

at any time during program development to see where the end of program is located in memory. The following example

```
EOP : EOD
```

could result in

```
@2D05
@2D55
```

This example indicates that the end of program is 2D05 and the end of data is at 2D55. EOD is valid only after data has been generated by running the program (after DIM has been executed).

FORMAT n

This statement sets the field size of printed numeric data, either a variable or number. FORMAT 0 is the initial setting. When the field is exceeded, n or n-1 asterisks are printed in the output for n > = 1. The minus sign is printed when the number is greater than the field in the negative direction.

```
***. . .* (number > format)
-**. . .* (number > format in negative direction)
```

The range of n is from 1 to 15.

LIST
LIST expr
LIST expr,expr

This statement (list the program), with no expression, will list the entire program in user space. If one expression is given, only one line, the number of which corresponds to the value of the expression, will be listed. Likewise, if two expressions are given and separated by commas, the listing will start and end at the respective lines corresponding to the values of the expressions. If, at any time, an expression evaluates to a number which is a nonexistent line number, then the nearest line number after will be listed.

NEW

This statement totally initializes BASIC3, the user space, and all data space. That is, it erases the user-generated BASIC3 program and initializes all string and array pointers. In addition, it automatically executes a CLS to clear the screen.

RENUMBER
RENUMBER n

The RENUMBER statement allows the user to renumber the lines within a program to a given increment. If no argument follows the RENUMBER statement, the increment defaults to 10, i.e., line numbers 10, 20, 30, and so forth. When n is given, it becomes the initial line number as well as the increment. The n value that is displayed for numbers above 256 will always be modulo 256. A warning message indicating the number of computed branches is issued. For example, consider the following program before and after a command "RENUMBER 20".

```
 1 DIM N(5)
 2 DEFINT C
10 FOR A=1 TO 5
15 LET N(A)=0
20 NEXT
25 GOTO B
```

becomes

```
20 DIM N(5)
40 DEFINT C
60 FOR A=1 TO 5
80 LET N(A)=0
100 NEXT
120 GOTO B
```

Further, a warning message is issued indicating the existence of 1 computed branch.

RUN

This statement sets BASIC3 into the program-execution mode. BASIC3 searches for the lowest line number and begins executing each line in numerical order. Before execution begins, however, the RUN statment clears all array and string data space to make room for new data.

RUN expr

This statement starts execution at a line number specified by expr. It is similar to RUN with no trailing expression in that it sets BASIC3 into the execution mode. If the line number specified by expr does not exist, an error code is generated. In addition, because this RUN statement does not clear the data space, the user can execute a program with data (strings or arrays) generated previously.

RUN+

This statement searches through the user's program and replaces interpretive branches with absolute address branches and then starts execution. This step greatly enhances speed. After the initial RUN+ cycle, RUN will cause execution in this faster mode. If any program editing occurs, the system automatically "UN-RUN+'s" the file.

A program converted by RUN+ should not be saved in this form because of the absolute address assignments.

TRACE expr

If the trailing expression evaluates to anything other than zero, the trace is turned on. That is, each line that is executed by BASIC3 will transmit the following to the output device:

```
TR[line number]
```

If a TRACE statement is executed with the trailing expression evaluating to zero, the trace will shut off. These

statements may be placed anywhere in a BASIC3 program. A very useful place for a TRACE statement is in an interrupt routine. An example of the TRACE statement is given under ENINT, DISINT.

Comment and Definition Statements

Comment and definition statements allow the programmer to insert comments into the program and to configure the memory, that is, to define the starting address of the program, to allocate memory for data, and the like. The comment and definition statements include DEFINT, DEFUS, DEG, DIM, FIXED, RAD, and REM.

DEFINT
DEFINT var name

This statement (define integer variables) when executed without any variable name sets all variables (A-Z) and all arrays (A(expr)-Z(expr)) as floating point. If a variable name is included in the DEFINT statement, all variables and arrays from A up to and including the variable name will be set up as integer. For example,

```
DEFINT D
```

defines variables and array names A, B, C, and D as integer. A NEW statement will always revert all variables to floating point. It is for this reason that, if integer variables are to be used in a program, a DEFINT statement should appear early in the program.

Note also that if a DEFINT is used with no variable name, it must end in a carriage return. That is, a colon may not be used to concatenate any further commands on the same line.

DEFUS expr

This statement (define the start of user space) is provided to allow the start of program space to be moved further back in memory. It allows the programmer to create a "hole" in memory in which he can generate machine language routines. Expr defines where the program space will begin. In BASIC3 the user space begins at hexadecimal 0200. The expression must evaluate to a number greater than 0200. BASIC3 will round down the expression to give only even-page increments of movement. If an attempt is made to define the user space at an address lower than 0200, BASIC3 will self-destruct. Once a DEFUS statement is executed, the only way to get the program space back to 0200 is by another DEFUS to 0200. The statement destroys the user program currently in memory.

An interesting feature of moving the start of user program space lies in the PSAVE statement. If a program has been generated at a moved location and it has associated with it some machine language routines, a PSAVE statement will save everything from 0200 to the end of program space. Included would be the machine language routines as well as the associated BASIC3 program. A subsequent PLOAD will load in all of the above, including the machine language routines, and will also redefine the start of user space to where it was when the file was created. No bookkeeping is necessary.

An example of the DEFUS statement follows.

```
DEFUS @2D00      or
DEFUS @2DCC      moves the start of user space
                 to 2D00
DEFUS @0200 moves the start of program space
            back to its original location
```

When the hole created by a DEFUS statement is used, it should be noted that memory location 0200 - 0210 is used in the saving routines and should be avoided when machine language routines are being written.

DEG

This statement sets the unit of measure for angular functions to degrees. The preset state of BASIC3 is radian measure. The complement of this statement is RAD.

DIM var list

This statement (dimension arrays) serves to reserve memory for arrays of numbers. These arrays may be one dimensional or two dimensional. The variable name (A-Z) is the name of one array. The expression defines how large the array is in one or both dimensions. See the earlier discussion of arrays and their size limitations. The mode of the array (floating point or integer) depends on whether its associated variable name has been defined as either integer or floating point. For instance, if all variables A-Z are floating point, then all arrays will be floating point (independent of the mode of the expression defining its limits). If variables A, B, and C are defined as integer, then arrays named A, B, and C (if used) will be defined as integer. Multiple arrays may be dimensioned in the same DIM statement so long as each array is separated by a comma.

If memory is exceeded when an array is being dimensioned, an error message will result. Trying to use any array element that has not been previously dimensioned will also result in an error message. The programmer must use a CLD statement to clear all data space.

The following is an example of a DIM statement.

```
DIM A(10,10),B(20)
```

The above example sets aside space for 100 numbers (10×10) with the names ranging from A(1,1) to A(10,10) and 20 numbers with names ranging from B(1) to B(20).

The dimension statement can be used in the direct mode of execution if desired. Any arrays generated by a program remain intact after program execution is completed. One can interrogate the contents of any array in the direct mode by means of the PRINT statement. The array remains intact until a CLD is executed or any editing to the program is done. Note that if a RUN expr is executed, the array has not been deleted and redimensioning will result in an error message. For that reason, the programmer should begin execution after the

DIM statement, if he wants to use previously generated data. The array data, as well as the string data, exists directly after the user space. Any editing of the original program will alter the user space and thus destroy the data. It should be noted that an array can be redimensioned without destroying other arrays or strings.

FIXED expr

This statement when executed formats the printing of all numbers both floating point and integer. The value of the expression defines how many digits to the right of the decimal point will be printed. Trailing zeroes will be filled in to complete the number. If necessary, the number will be rounded. The expression must evaluate to a number between 0 and 6. If a number greater than 6 is entered, BASIC3 reverts to its normal mode of outputting numbers. For example,

```
FIXED 2
PR 123      will result in 123.00
PR 123.567  will result in 123.57
PR 6E7      will result in .60E08
FIXED 7
PR 123.50   will result in 123.5
```

RAD

This statement sets the unit of measure for angular functions to radians. This state is the preset state of BASIC3. The complement of this statement is DEG.

REM

When REM is placed at the start of any BASIC3 line, the entire line is listed but ignored during program execution. Its purpose is to enable the programmer to document the listing.

Assignment Statements

Assignment statements assign values to register R0, variables, memory locations, or the Q flag. The assignment statements include DMAPT, LET, POKE, and STQ.

DMAPT expr

This statement provides a means to load R0 (register 0, the DMA pointer) with some value defined by expr. For example,

```
DMAPT @2E23
```

would load 2E into R0.1 and 23 into R0.0

LET var name = expr

This statement assigns the value of the expr to the var name. The mode of the variable name defines the mode in which the expression will be evaluated. The LET keyword is optional and may be omitted. The following are examples of acceptable LET statements.

```
LET A=1
B=2*A
LET K(B)=SIN(A+2)
R(5)=2*K(B)
```

LET string var = string expr

This statement is exactly the same as the previous LET statement.

Again, the LET keyword is optional. This LET statement serves to assign a set of alphanumeric characters to a string variable. Some examples are as follows.

```
LET A$="HELLO"
B$=A$+"GOODBYE"
A$(5)=MID$(A$,1,3)
LET B$(2)=A$+A$(1)
C$(10)=B$(N)+A$(5*3/A)+ + C$
```

Note that in the LET statement all strings of characters must be enclosed in quotes.

POKE (expr1,expr2)

This statement, which should be used with extreme caution, is used to write any location in memory with any data. The location in memory is defined by expr1 and the data is defined by expr2. For example,

```
POKE (5000,255)
```

will place the hexadecimal equivalent of 255 (FF) in memory at the decimal address 5000. A more common and much safer approach is

```
POKE (@1F23,#FF)
```

which places the hexadecimal data FF at hexadecimal location 1F23. See the definitions of @ and # in the operator descriptions in the preceding chapter. Refer also to PEEK in Chapter 3, **Functions.**

STQ expr

This statement sets Q to either 0 or 1. The expression may be any value. BASIC3 looks only at the least significant bit of the rounded whole number.

Control Statements

Control statements control the flow of a program by performing conditional and unconditional transfers. The control statements include END, EXIT, FOR, GOSUB, GOTO, IF, NEXT, RETURN, and WAIT.

END

This statement (end of program) serves as a stop or end. It terminates program execution and returns BASIC3 to the direct execution mode. An END statement may be placed anywhere and any number of times in a BASIC3 program. It may also be deleted completely if desired as the last statement in a program.

EXIT expr

This statement is an unconditional branch to a line number defined by expr. It is intended for a premature es-

cape from a FOR/NEXT loop or a subroutine. One precaution should be noted. If subroutines or FOR/NEXT loops are nested, the EXIT statement is designed to transfer control to a line number within the next level down of nesting. For example, if FOR/NEXT loops are nested four deep and it is desired to prematurely branch outside of the fourth FOR/NEXT loop, BASIC3 expects to be somewhere in the third FOR/NEXT loop after the EXIT has been executed. An EXIT may then be taken to some line within the second FOR/NEXT loop, and so on. This statement is used instead of the standard GOTO and is designed to clean up all of the modified stack pointers that result from a FOR/NEXT or a subroutine call. An example follows.

```
10 FOR I=1 TO 10
20 FOR J=1 TO 5
30 A=B+C
40 IF A=15 THEN EXIT 60
50 NEXT J
60 PR A
70 NEXT I
```

FOR var = expr1 TO expr2 STEP expr3

This statement assigns to a variable the value of expr1. The program continues executing until a NEXT statement is encountered. At that time the value of the variable is incremented by the amount defined by expr3 which may be either positive or negative. Expr2 - var is then evaluated and compared to the sign of the step. It is assumed that zero has a positive sign. If a match in sign does not occur, execution resumes at the first statement after the last FOR statement and continues through the loop. At any time during the loop, the user can modify the value of the variable name by any available means (e.g., a LET statement). It should also be noted that the mode of the variable name sets the mode in which expr1, expr2, and expr3 are evaluated. If the STEP is deleted, a step size of one is assumed.

The following are examples of the FOR/NEXT loop.

```
10 FOR A=1 TO 3 : FOR B=1 TO 6 STEP 2
20 PR A,B
30 NEXT B : NEXT A
40 PR "DONE"
```

This example would result in:

```
1     1
1     3
1     5
2     1
2     3
2     5
3     1
3     3
3     5
DONE
```

An example of a negative step follows.

```
10 FOR A=10 TO 6 STEP -1
20 PR A
30 NEXT
40 PR "DONE"
```

This example would result in

```
10
 9
 8
 7
 6
DONE
```

A third example follows.

```
10 FOR A=1 TO 10
20 IF A=5 LET A=9
30 PR A
40 NEXT A
50 PR "DONE"
```

This example would result in:

```
 1
 2
 3
 4
 9
10
DONE
```

GOSUB expr

This statement (subroutine call) is identical to the GOTO statement with the exception that the program remembers where the GOSUB occurred. When BASIC3 encounters a RETURN statement, it will return to the statement following the last GOSUB statement executed. In this way subroutines may be nested as many deep as memory will allow (the stack grows as the number of nested GOSUB's grows). An example of the GOSUB statement is

```
5 A=2000
10 GOSUB 1000 : GOSUB 1000
20 GOSUB A
30 END
1000 PR "DONE --"
1010 RETURN
2000 PR "FINISHED":GOSUB 3000 :GOTO 1010
3000 PR "COMPLETE" : RETURN
```

This example results in the following being printed.

```
DONE --
DONE --
FINISHED
COMPLETE
```

GOTO expr

This statement is an unconditional branch. It transfers execution immediately to the start of the line number specified by the expr. If the line number does not exist, an error message is generated. Examples are as follows.

```
10 GOTO 50
10 GOTO A+B
10 GOTO 100*(A-B)
```

IF string reference <> string expr THEN statement
IF expr (relational operator) expr THEN statement

This statement is a conditional statement but is not conditional branch as in some other Basic's. It tests for a condition, and if the condition is met, a statement or group of statements (separated by colons) is executed. If the condition is not met, then execution continues at the next line number. When string relations are compared, an equals or not equals sign is the only acceptable relation. In the case of arithmetic expressions, the following are acceptable relational operators.

=	equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

The mode of the first expression defines the mode of the second expression. An example of a conditional statement follows.

```
10 IF A=B THEN PR A:WAIT(200):CLS:GOTO 200
20 PR B
```

In this case, if the value of A is equal to the value of B, then the value of A will be printed, a delay will occur, the screen will be cleared, and execution will continue at line #200. If A is not equal to B, then the value of B is printed and execution continues from line #20 onward.

One additional point to be made is that the keyword THEN is optional and need not be used. A common mistake to be avoided is the following.

```
IF A>N THEN 200
```

Anything to the right of THEN, if it appears, is to be an executable statement. The number 200 is not an executable statement. The correct version would be as follows.

```
IF A>B THEN GOTO 200 or
IF A>B GOTO 200
```

Some more examples of acceptable IF statements are given in Fig. 3. Note the multiple conditions in the last example. If any of the conditions are not met, execution continues at the next line number. If all of the conditions are met, execution will finally get transferred to line #500 by means of the GOTO statement. It follows that the GOTO statement could have been any executable statement.

NEXT
NEXT simple var

This statement closes the FOR/NEXT loop as described in the FOR statement. If the variable name is omitted, the NEXT statement returns to the last FOR statement and continues. If the variable name is included, BASIC3 will check to see if it matches the variable name used in the last FOR statement. If it does not match, an error message is issued.

RETURN

This statement (return from a subroutine call) marks the end of a subroutine. Execution returns to the statement following the last GOSUB executed.

WAIT (expr)

This statement provides a way to insert a delay in the execution of a program. The length of the delay is directly proportional to the value of the expression. 1 equals 16000 CPU clocks. At a 2-megahertz clock rate, the delay for an expression of 1 is approximately 8 milliseconds (i.e., $16000 \times 1/(2 \times 10^6)$). An example of one application of the WAIT statement is as follows:

```
10 INPUT A$(1)
20 PR "MESSAGE IS";A$(1)
30 WAIT(500) : CLS : GOTO 10
```

This routine allows the message to be viewed for some period of time (500 x 8 ms) before the screen is cleared and execution continued.

Program Data Statements

Program data statements allow the programmer to embed lists of data in the program and provide the mechanism to read the data during program execution. The program data statements include DATA, READ, and RESTORE.

DATA date list

The DATA statement contains data to be used by a READ statement. Each element in the DATA statement must be separated by a comma. Any string of characters

```
10 IF A(2)*B(1)>=C*SIN(A) PR A : GOTO 50
10 IF A$(2);nn"LIST" LET B=3 : GOTO 100
10 IF B$(A+B)=MID$(A$,2,4) PR "OK"
10 IF MID$(A$(5),2,4)="EBCD" PR"MATCH" : GOTO 500
10 IF A$=B$+C$ THEN GOSUB 200 : CLS : PR"DONE"
10 IF A=INT(S/5) IF B>10 IF C<5 GOTO 500
```

Fig. 3 - Examples of acceptable IF statements.

must be enclosed in quotes. The DATA statement may appear anywhere in a BASIC3 program. The following are acceptable DATA statements.

```
    1 DATA 2*A,A$(1),"HELLO",B
 1000 DATA 1,2,3,4
 5000 DATA SIN(45),SIN(60),SIN(90)
```

READ var list

The READ statement is used to read data from the DATA statement and assign that data to a particular variable. Each time the READ statement requests more data, an internal BASIC3 pointer moves to the next piece of data in the DATA statement. When a DATA statement is out of data, BASIC3 will search onward for another DATA statement. If none is found, an error message is returned indicating the lack of data. It is important to keep track of the variable names or string names in the READ statement and the corresponding data in the DATA statement. They must match in form; strings go with strings, and so on. An example of an acceptable DATA/READ statement pair is as follows.

```
5  DIM A(4)
10 DATA 10,20,30,40
20 FOR A=1 TO 4
30 READ A(A)
40 NEXT
50 READ A$(1),B,C
60 PR A$(1);B*C
70 DATA "B*C IS EQUAL TO",20,30
```

The result of the above example is that the array A would be loaded as follows:

```
A(1)=10 A(2)=20 A(3)=30 A(4)=40
```

Line # 60 would result in

```
B*C IS EQUAL TO 600
```

RESTORE

This statement resets the previously mentioned BASIC3 pointer back to the start of the first DATA statement in the program, thus allowing the multiple use of DATA in one program.

I/O Statements

I/O (input/output) statements provide the interface between a program and the program user. The I/O statements include INPUT, OUT, PRINT or PR, TIN and TOUT.

INPUT var list

When BASIC3 encounters an input statement, it stops and issues a question mark to the terminal or disk as a prompt. It then waits for a programmer response. A list of variable names may appear after the INPUT statement. Each of the variable names must be separated by a comma. In response to the BASIC3 prompt (?), the programmer may answer with any expression or group of expressions separated by commas. If the INPUT statement has three variable names after it and the programmer responds with only two expressions, BASIC3 will issue another prompt (?) and wait for the third expression. Conversely, if the INPUT statement has only one variable name and the programmer responds with two or more expressions, BASIC3 will continue execution. When it encounters another INPUT statement it will pick up the last unused expression without issuing another prompt. It will continue to do so until all the unused inputted data is used up. Then, at that time, another prompt (?) will be issued. The data that is picked up is assigned to the associated variable name as it is inputted. The mode of the variable name defines the mode in which the associated inputted expression will be evaluated.

Note that an INPUT statement may not contain both string variable names and normal variable names at the same time. Furthermore, each INPUT statement containing a string variable name can contain one and only one name. Because of this limit, in response to an INPUT A$ statement, the programmer can answer with a string of characters with no quotation marks. The carriage return marks the end of the string. The following are acceptable INPUT statements.

```
10 INPUT A,B,C,(1)
10 INPUT A(1) ,A(B)
10 INPUT A$
10 INPUT B$(B)
```

The following are incorrect INPUT statements.

```
10 INPUT AB (no delimiter between variables)
10 INPUT A$,B$ (multiple string names forbidden)
```

INPUT "string" var list

This INPUT statement is identical with the one above except that directly after the keyword INPUT a quote appears allowing a message to be printed prior to the question mark prompt. No delimiter is used before the first quote or after the last.

OUT(expr1,expr2,expr3)

This statement outputs the value of expr3 to the output port defined by expr1 and expr2. Expr1 defines the group number and expr2 defines the port within the group. (For a discussion of two-level I/O see the instruction booklet MB-601 for the COSMAC Microboard Computer CDP18S601.) The data defined in expr3 should evaluate to a number less than 256 in order to have it expressed in eight binary bits (the port size). Refer also to INP in Chapter 3,**Functions.** An example follows.

```
OUT (10,2,4 OR \10010101\)
```

If expr1 evaluates to 0 and expr2 evaluates to 1, a group select is performed. An example follows.

```
OUT (0,1,2)
```

selects group 2.

Any time a group select is performed, its system stays in the group. If the user wishes to remain in group 1 after a device in group 2 is ad-

dressed, an OUT(0,1,1) must be executed. An example follows.

```
OUT(2,2,2)
OUT(0,1,1)        (return to group 1)
```

PRINT or PR
PRINT print list

PR is used as an abbreviation for PRINT. In a listing PR's are replaced with PRINT's. The purpose of this statement is to output to the I/O routines any expr or string function as defined earlier. Each of the items to be printed must be separated by an acceptable delimiter, either a comma or a semicolon. Each of the delimiters serves a special function. When a comma is used, the next item to be printed is placed at the next eighth column increment. When a semicolon is used, no spaces are inserted and the next item is printed directly after the last. The semicolon and comma also serve to inhibit the carriage return at the end of PRINT statement. The next PRINT statement encountered in the program, therefore, will begin where the previous one left off. If a PRINT statement is used all by itself, then a carriage return/line feed is outputted. There are two functions that are usable only in a PRINT statement: TAB(expr) and CHR$(expr). A description of these two functions is given in the next section. Examples of acceptable PR statements are as follows:

```
PRINT 5
PRINT A
PR (A+B)/C
PR A,B, 1234,C(5)
PR "THE VALUE OF A= ";A
PR A$(1)+A$(2)
PR A$(2),A,B;
PRINT MID$(A$,1,2)
PRINT TAB(A+B);10;TAB(30);E
PR A,
PR
```

Note that when more than one arithmetic expression appears in a PRINT statement, each expression may be evaluated in a different mode.

Saving or Loading a Program or Data on Tape Statements

These statements allow a program to treat the tape as bulk memory for the storage and retrieval of bulk data. Statements include DLOAD, DSAVE, PLOAD, and PSAVE.

PLOAD n

PLOAD (program load) loads a program from tape n (n = 0 or 1) to user space. When the PLOAD statement is executed, the entire user space including data is cleared and replaced with a previously saved program (generated by PSAVE).

PSAVE n

This statement (program save) stores memory from #0200 to the end of the user program on tape at tape drive n (n = 0 or 1).

DSAVE n

The DSAVE statement (data save) is similar to PSAVE except that the memory saved on tape begins at the end of the program (start of data) and ends at the end of data. This statement provides the capability of saving any program-generated data (strings and/or arrays).

DLOAD n

The DLOAD statement (data load) loads from tape drive (n = 0 or 1) any previously generated data (from a DSAVE). The data may be loaded with any BASIC3 program. The data is automatically placed at the end of existing program space and over writes any existing data.

Caution: Three points must be considered with DLOAD.

1. Because a PLOAD wipes out previous data and then loads the program, a DLOAD must be executed after a program is in user space. Any editing of the user space will also wipe out data. Therefore, if editing is necessary, a DSAVE should be executed to save the data (if desired), the program edited and a DLOAD executed to return the data.
2. The act of doing a DLOAD automatically dimensions any arrays within the stored data.
3. Strings are placed at the end of array space. If additional arrays are dimensioned after strings have been generated, the strings will be wiped out by the growing array space. For this reason, all array dimensioning should be done before any strings are generated for permanent storage.

It should be noted that if a program and its data are to be stored, the following can be executed (n = 1 or 0).

```
PSAVE n
DSAVE n
```

To execute the program, execute the following.

```
RUN expr
```

where expr is the line number beyond any dimension statements.

Sequential Device I/O Statements

Sequential Device I/O statements allow a program to address allowable peripheral devices for either output or input of sequential data. Allowable input devices are the tape and the terminal keyboard. Allowable output devices are the tape and terminal display. Statements are CLOSE, DIN, DOUT, RFLN, TIN, TOUT, and WFLN.

RFLN
WFLN

These statements are used to open a read or write tape drive, respectively. RFLN 0 reads from drive 0; WFLN 1 writes to drive 1. Default is drive 0. Only one read tape and one write tape may be opened at any time. Opening a second read tape would result in the abandoning of the first one. Opening a second write tape automatically closes the first write tape. RFLN permits the user to call in overlays. Overlays provide a powerful program-segmentation technique when memory space is limited.

DIN
DOUT

These statements set the tape drive as the input or output device, respectively, instead of the terminal. All data normally outputted to the terminal would, instead, be put out on the tape under the file name and tape drive number defined in the last WFLN statement. All input response will come from tape under file name and tape drive number defined in the last RFLN statement. Care must be exercised to make sure that once control has been given to the tape as the input device, an "END OF FILE" message is outputted to the current output device. If the current output device is the tape, the prompt would go unnoticed and the terminal is automatically redefined as the input device.

TIN
TOUT

These statements set the terminal as the input or output device, respectively. See DIN/DOUT for a further discussion.

CLOSE

This statement, when executed, closes the current write file by placing a DC3 at the end of the file and emptying the remainder of the buffer to the tape. A NEW instruction, when executed, also does an automatic CLOSE to the write file. Any generated BASIC3 error message also automatically executes a CLOSE.

Machine Language Subroutine Statement

Machine language subroutine statements allow the programmer to include machine language code with his program to take advantage of particular characteristics of the processor or for speed enhancements in real-time applications. The machine language subroutine statement is CALL.

CALL (expr1)
CALL (expr1, expr2)
CALL (expr1, expr2, expr3)

This statement provides the link between BASIC3 and machine language programming. It serves as a machine language subroutine call. It transfers execution to a machine language subroutine, the address of which is determined by expr1. The machine language routine should be written with the following rules in mind.

1. The program counter upon entry into the subroutine is R3.
2. Transfer is made back to BASIC3 by means of a D5 (SEP R5) instruction.
3. The machine language routines have free use of R8, RA, RC, RD, and RE. If any other registers are to be used, they should be saved first on the stack and restored before returning to BASIC3.

   Note: The standard call and return technique (SCRT) is not used because it destroys the upper part of register F (RF.1).

4. Call and return conventions have been established by BASIC3 and no further initialization is required by the machine language subroutines.
5. The stack is available for use (pointed to by R2) so long as it returns as it was left.
6. If a UT62 TYPE or READ routine is called in the user subroutine, register C must be set up to point at the DELAY ROUTINE in UT62. This address is #80EF. The DELAY routine is used by TYPE and READ in conjunction with the software UART in UT62.

Any of the expressions (expr1, expr2, expr3) may be expressed in either integer or floating point. BASIC3 will automatically convert them to integer. The value of expr2, if used, is then passed to the machine language subroutine as a 16-bit binary integer number in R8. A second piece of data may also be passed to the machine language subroutine in register RA. The value will be that of expr3. RD is initialized to timing constant for utility program.

3. Functions

This chapter contains a description of the functions provided in BASIC3. The functions are divided into four groups and are discussed in alphabetical order under each group head. The four group heads are:

Arithmetic functions
String functions
I/O functions
Machine language functions

Arithmetic Functions

The functions in this group include ABS, ATN, COS, EXP, FNUM, INT, INUM, LOG, MOD, PI, RND, SGN, SIN, and SQR.

ABS (expr)

This function returns the absolute value of the expression. For example

```
PR ABS (-10*2)
```

will print

```
20
```

ATN (expr)

This function returns the angle (expressed in radians) whose tangent is equal to the value of the expression. This function is a floating-point function.

COS (expr)

This function returns the cosine of the angle determined by the value of the expression (expressed in radians). This function is a floating-point function.

EXP (expr)

This function returns a floating-point number of 2.71828 (e) raised to the power of the value of the expression.

FNUM (expr)

This function rounds the expression to the nearest whole number and converts it to the floating-point mode. It is an exact counterpart to INUM. For example.

```
B=FNUM(RND(6) ):PRB
```

where B is not an integer - will result in a floating-point number between 0 and 5. would result in

```
6
```

INT (expr)

This function returns the integer part of the floating-point expression truncating out the fractional part. The result is still in the floating-point mode. This function is not to be confused with INUM (described later).

INUM (expr)

This function converts the floating-point expression to the integer mode, rounding to the nearest whole number. It provides a means for forcing a particular function to the integer mode. For example,

```
PR SQR(50)
```

would result in

```
7.07107
```

and

```
PR INUM (SQR(50) )
```

would result in

```
7
```

LOG (expr)

This function returns a floating point number the value of which is the natural logarithm of the value of the expression.

MOD (expr1, expr2)

This function (modulo) returns as its value the following equivalent

```
expr1 - (expr1/expr2) expr2
```

where each expression is first converted to integer. The

function defaults to an integer mode value. for example

```
PR MOD(10,3)
```

would result in

```
1
```

PI

This function returns as its value 3.14159.

RND

This function returns a random floating-point number greater than or equal to zero and less than or equal to one.

RND (expr)

This function returns a random integer number greater than or equal to zero and less than the value of the expression.

SGN (expr)

The value of this function is either +1, 0, or −1 depending upon the sign of expr. If expr is positive, the function returns a +1. If expr evaluates to zero, the function returns as its value a 0. If expr is negative, the function returns a −1.

SIN (expr)

This function returns the sine of the angle determined by the value of the expression (expressed in radians). This function is a floating-point function.

SQR (expr)

This function returns the square root of the value of the expression. This function is also floating point.

Examples of the above functions are as follows.

```
PR SIN(45)
A=INT(10*SQR(LOG(A*B) ) )
C=EXP(10+A)
PR ATN(A*SIN(A) )
```

String Functions

The function in this group include ASC, FVAL, CHR$, LEN, and MID$.

ASC (string expr)

This function returns as its value the decimal equivalent of the ASCII value of the first character in the referenced string. The value returns in floating point as a default. If it has been preceded by any integer function it will be converted to integer. For example

```
A$(5)="ARITHMETIC"
B=ASC(A$(5) )
```

The variable B would contain as its value, 65., the decimal ASCII value of the letter A.

As another example

```
5  A$="TEST"
10 FOR A=1 TO LEN(A$)
20 PR ASC(MID$(A$,A) )
30 NEXT A
```

would result in

```
84
69
83
84
```

CHR$ (expr,expr,. . . .)

This function is used and recognized only in the PRINT statement. It evaluates the expression and outputs the corresponding ASCII decimal equivalent to the I/O routines. For example

```
PR CHR$ (65)
```

would result in

```
A
```

and

```
PR CHR$ (#42)
```

would result in

```
B
```

and

```
PR CHR$ (#41,#42,#43)
```

would result in

```
ABC
```

FVAL (string expr)

This function evaluates the string expression as an arithmetic expression and returns its value. In this manner the user can create mathematical functions. The following contain three examples of FVAL usage.

```
10 A$="8+4"
20 PR FVAL (A$)
30 PR FVAL (A$+"/2")
40 PR FVAL ("SQR(3)")

:RUN (execute program)

 12       (evaluate and print A$)
 10       (evaluate and print FVAL ("8+4/2") )
 1.73205  (evaluate and print SQR (3) )


 READY
```

The string is limited to about 48 characters. Beyond that length, valuable data area may be clobbered.

LEN (string var)

This function returns as its value the number of characters in the specified string. The value returns in floating point unless it is preceded by an integer number or function, in which case it is automatically converted to integer. The following is an example.

```
A$="ARITHMETIC"
PR LEN(A$)
```
gives
    10 (the length of A$)
and
```
PR A*LEN(A$)
```
gives
    10 (assumes the variable A was defined as integer with a one stored as its value)

Note that string must be previously defined.

MID$ (string var, expr1)
MID$ (string var, expr1, expr2)

This function is a string function which, when executed, extracts a portion of the specified string. The first term, expr1, defines which character from the left is to start the substring. Expr2 defines the number of characters to be used in the substring. If expr2 is not used, all of the remaining characters are used. The following is an example.

```
A$="YES"
B$=MID$(A$,2,1)
```
B$ would then contain the letter "E".
```
A$(10)="EXPERIMENT"
PR MID$(A$(10),7,3)
```
would result in
    "MEN"
being printed.
```
10 INPUT A$(1)
20 IF MID$(A$(1),1,1)="Y" GOTO 100
30 GOTO 10
```
The last example would wait until some word beginning with the letter "Y" was inputted at which time the program would branch to line #100.

Common functions like LEFT$ and RIGHT$ of other BASIC's can be implemented with the MID$ function in the following way
```
MID$ (A$,1,1)
```
is the same as
```
LEFT$ (A$)
```
and
```
MID$ (A$,LEN(A$) )
```
is the same as
```
RIGHT$(A$)
```

I/O Functions

The functions in this group include EF, INP, MEM, PEEK, QST, and TAB.

EF(expr)

This function returns the state (0 = false, 1 = true) of the CPU flag (expr) where expr = 1 through 4. The expression cannot be greater than 4, but it can be either floating point or integer. It returns a 0 or 1.

INP (expr1,expr2)

The value of this function is the decimal equivalent of the 8-bit port which it calls. Expr1 defines the group number and expr2 defines the port within the group. Once a group has been selected, the system stays in that group. To return to another group, the user must execute an OUT(0,1, Group). Refer also to OUT in Chapter 2, **Statements.**

MEM

This function enables the user to determine how much memory is left. When executed, MEM returns a decimal number representing the number of memory bytes left between the end of data (strings and arrays) and the end of the normal stack. The actual number returned is reduced by 256 to allow for stack growth during program execution. Note that if MEM returns a negative number, it means that the end of data is within the 256 bytes of margin and caution should be used in allowing the program to grow any further. No provision is made to account for the stack growing into the program during program execution. To receive the value on the screen the user should type
```
PR MEM
```

PEEK (expr)

This function returns the decimal equivalent of the contents of memory at an address determined by the expression. The decimal result defaults to floating point. Refer also to POKE in Chapter 2, **Statements.**

QST

This function queries the Q line. It tests for a 1 (SET) or 0 (RESET) and returns the value.

TAB (expr)

This function is not a function in the same sense as the previous ones in that it does not return a value of any kind. It is used and recognized only in the PRINT statement. It places the cursor at the horizontal position determined by the value of the expression. The new cursor position is referenced to column 0 and not to the previous cursor position. Printing continues from that point on. Following are some examples.
```
PR TAB(10);1
```
will print a 1 at column 10.
```
PR TAB(10);1;TAB(20);2
```
will print a 1 at column 10 and a 2 at column 20.
```
PR TAB(A);"*"
```
will print a star (*) at column A.

Machine Language Function

The function in this group is USR.

USR (expr1)
USR (expr1,expr2)
USR (expr1,expr2,expr3)

This function acts like the CALL statement described in the previous section but with the difference that USR is a function to be used as part of an expression. When USR is encountered, a subroutine call is made to the machine language routine stored at expr1. Data may be passed to the subroutine in exactly the same way as the CALL statement. With the USR function, when a D5 is encountered in the machine language routine, BASIC3 will return a 32-bit binary integer number as a value for the USR function. This 32-bit number will be constructed from R8 and RA. R8 provides the low-order 16 bits and RA provides the high-order 16 bits. RD is initialized to timing constant for utility program. An example follows.

```
PR 2*USR(@3C1E,#2F)+A/D
```

Note that if the USR subroutine calls UT62 READ or TYPE routines, RC must be initialized to point at #80EF (DELAY) as in BASIC3 CALL statement.

4. Creating and Editing Programs

BASIC3 programs can be created and edited by either of two methods. One is by use of the BASIC3 interpreter; the other is by use of the MCDS Resident Editor (CRE).

When the interpreter is used, a program is created by the writing of one or more statements, separated by colons, on a line and assigning the line a line number. The BASIC3 interpreter allows the lines to be entered in any order but for execution it will automatically rearrange them in numerical sequence. For example, line 10 may be entered before line 5, but in execution line 5 will be executed first. This facility enables the programmer to leave unused numbers between lines so that additional lines can be inserted at a later time. The interpreter always executes the lines in numerical order starting with the lowest line number, thus providing one method of editing a program.

If a sequence of numbers is fully used and it becomes necessary to insert a line, the programmer may use the RENUMBER command (see Chapter 3, **Statements**) to create a space. The deletion of a line is accomplished by typing the line number followed by a carriage return. Issuing a LIST command will display the program with the lines in the proper sequence. A line may be edited on entry by use of the rubout (delete) key. Use of this key causes the elimination of the previous character and, at the same time, echoes it back to the terminal. Backspace (CONTROL H) deletes characters from the screen and line buffer. The CONTROL C key has the effect of deleting the entire line during entry.

The EDIT command statement, described in Chapter 2, **Statements,**, also provides for character change, insertion, and deletion.

Programs created via the interpreter can be saved on disk and loaded from disk by use of the PSAVE and PLOAD commands, respectively. These commands are also discussed in Chapter 2, **Statements.**

The second method of creating and editing programs is by use of the MCDS Resident Editor. This method is described in detail in the **User Manual for the RCA COSMAC Microboard Computer Development Systems (MCDS) CDP18S693 and CDP18S694,** MPM-293. Following is a simple example of such a programming session.

1. Run the editor by loading it through the UT62 monitor by typing "E" and generate the following file.

```
10 INPUT "WRITE TAPE"Z
15 INPUT "READ TAPE"Y
20 WFLN Z
30 RFLN Y
40 DOUT
50 FOR A=1 TO 10
60 PR A
70 NEXT A
80 CLOSE
90 TOUT
100 PR "DATA NOW INPUT FROM WSC"
110 DIN
120 FOR A=1 TO 10
130 INPUT C : PR C
140 NEXT A
150 TIN
160 END
 TIN
```

2. Write that file on to tape 0 or 1.
3. Leave the editor and call BASIC3.
4. When the READY appears, type the following:
   RFLN 0 or 1.

Note that the drive number 0 is defaulted. BASIC3 will open the read tape and return with READY.

Next, type the following:

```
DIN
```

At this point, BASIC3 will ignore the terminal and receive its input from the tape file. The tape contains a series of lines with numbers to be entered as a program into BASIC3. The last line read from the tape by BASIC3 is a TIN, which is directly executable because no line number was used. This statement has the effect of returning control to the terminal and a READY will be issued.

Type LIST from the terminal and the stored BASIC3 program will be displayed.

Next, type RUN and the following will occur. The program will prompt the user for a tape drive in which to store data. The write tape will be opened in line #20. Line #30 opens a read tape. Line #40 sets the tape as the output device and all output will be directed to tape drive Z. Lines #50, #60, and #70 generate a list of numbers from one to ten in a format that will be acceptable as INPUT data. Line #80 closes the write tape, and with line #90 the terminal is returned as the input device. A message is printed in line #100, and the tape is set as the input device in line #110. Line #120 begins a FOR/NEXT loop that contains an INPUT statement. Because the tape is the input device, BASIC3 will look to the read tape as the response to the INPUT. Because a column of numbers was entered into the tape earlier by means of the PRINT statement, the numbers are used as input data during the INPUT and printed in line #130. At line #150 the terminal is returned as the input device and the program is halted at line #160. If the user were to leave BASIC3 by typing BYE, the write tape could be listed through the editor in the MCDS.

The above example demonstrates data saving and retrieval by means of the tape. The possibilities for mass data storage and retrieval are endless. The above example also demonstrates the use of the EDITOR and all of its capabilities as a means of generating, modifying, or correcting a BASIC3 program. These same techniques can be used to save ASCII files on a tape for later editing or listing to a line printer. The command sequence, in the immediate mode, would be as follows:

:WFLN 0 or 1 :DOUT:LIST:CLOSE:TOUT

One additional comment could be added regarding the above example. At any time after the program had been entered, it could have been saved in its compressed form with use of a PSAVE statement. See the description of PSAVE and PLOAD in Chapter 2, **Statements.**

5. Error Messages and Program Debugging

Whenever the BASIC3 interpreter detects an error in a statement, it generates an error message consisting of ERR CODE Z and a two-digit decimal number followed by the message READY and the : prompt symbol. A listing of the error numbers and their corresponding meaning is given in Appendix A. If the error is detected during program execution, the error code is followed by the words AT LINE followed by the line number of the offending statement.

Example

```
ERR CODE 07
  AT LINE 130
  READY
  :
```

Most program execution errors are due to either incorrect flow or improper modification of variable values. To find an error of the first kind, the user must determine whether the program is sequencing properly, i.e. whether certain sections of code are indeed executed when expected. Often, the insertion of dummy PRINT statements within suspected code sections will reveal whether the flow within the program is proper.

The second type of error is most easily detected by the insertion of dummy program stops at key points. This procedure is also useful for diagnosing incorrect flow. A dummy stop is an inserted END or some other inserted statement that is intentionally erroneous to cause an error stop. Once the stop occurs, the value of key variables may be examined (using the immediate execution mode - e.g., PRINT A, B, C) to see if they have the expected behavior. In some cases, variable values may be corrected in the immediate mode while the program is still stopped. In such cases and in the case where the program behavior is correct so far, the user will want to resume the program at the point where it last stopped. An immediate or direct GOTO, using the statement number after the stop, will permit the program to proceed as if it had not been interrupted.

One warning is issued by BASIC3. The warning is <<MEM WARN>>. This warning may be encountered as a BASIC3 program is being entered. The warning is issued when the end of program space grows to within 256 bytes of the end of the stack. The typing of PR MEM at this time will return a negative number indicating how far into the margin area BASIC3 has grown.

The TRACE statement of BASIC3 is a useful debugging tool. It permits the user to selectively view a display of line numbers giving the sequence in which statements are being written.

6. Run-time BASIC CDP18S842

Run-time BASIC CDP18S842 is a variant of BASIC2 CDP18S840 and BASIC3 CDP18S841. It is available in ROM and is especially suited for custom Microboard applications not requiring disk I/O. The Run-time BASIC interpreter is located in memory from 0000H to 2FFFH (hexadecimal notation). The interpreter starts execution of a user program at 3000H in ROM and generates data in RAM at 9000H. There must be at least 1 kilobyte of RAM available at 9000H for the interpreter to run. The work pages for Run-time BASIC are at 9000H and 9100H. Any references to the work pages in the user program should be to these addresses and not to the ones for BASIC3.

The primary difference between BASIC3 and Run-time BASIC, other than that there is no program input mode for Run-time BASIC, is that the latter cannot execute PLOAD or PSAVE. All other instructions are valid. The statements WFLN, RFLN, DIN, and DOUT, in Run-time BASIC apply to a tape I/O system such as the Microboard Computer Development System MCDS CDP18S693 or CDP18S694. The CDP18S652 Combination Memory and Tape I/O Control Microboard is required for tape I/O with Run-time BASIC.

Using Run-time BASIC

To generate a Run-time BASIC program from a BASIC3 program proceed as follows:

1. Save the BASIC program on tape in ASCII.
2. Determine the RAM requirements for the program.
3. Make a ROM or EPROM containing the program for use with the Run-time interpreter.

Two examples of this conversion technique are given below:

Example 1:

1. Use the EOP command in BASIC3 to find the length of the BASIC3 program (0200H to EOP).
2. Use the EOD command in BASIC3 to establish the RAM requirements. The value difference between the EOP and EOD command results, plus two work pages required by Run-time BASIC, determine the total RAM requirements.
3. Save the BASIC program on tape by returning to the UT62 Monitor program with the BYE command.
4. Use the BASIC3 program on tape with the PROM programmer option of the MCDS CDP18S694 to make a Run-time EPROM directly or, list the program area on the terminal for use in a ROM or in an EPROM generation system.

Example 2:

1. Pass a PSAVE'd tape through a program that converts "pseudocode" into ASCII as used in the RCA COSMAC Development System (CDOS) CDP18S008. Such a program is given in Appendix I. This program can be edited and assembled with the MCDS CDP18S694 Resident Assembler/Editor.
2. Start the program from the UT62 Monitor with "P0" which prompts the user to mount tapes.
3. The tapes thus produced can then be ROM'ed or PROM'ed.

The Monitor program UT62 is only required with tape I/O for mass storage of data. READ and TYPE routines have been supplied in Run-time BASIC in both UART and in software UART ("bit banger") forms. The default is to the bit-banger routine at 300 baud. If the UART routines are required, the first statement of the BASIC program should POKE the address of the UART routines into the work page. See Appendix C for the addresses of the routines and the bytes in the work page to be changed. Examples of POKE statements are given in Fig. 4. The echo flag/baud rate is the software UART's baud rate. This rate can be modified if required. The following cal-

culations for the Timing Parameter (TP) can be used.

$$TP = 2 * \frac{\text{interval between two serial bits}}{320 * (\text{CPU clock period})}$$

where the fraction is rounded off to the nearest integer. If the user is calculating for 110 baud at a CPU clock rate of 2.0 MHz, the calculation would be as follows.

$$TP = 2 * \left[\frac{\frac{1\text{ s}}{10\text{ char}} \quad \frac{1\text{ char}}{11\text{ bits}}}{320 * \frac{1\text{ s}}{2.0 * 10^6}} \right]$$

$$TP = 2 * 56.8 \text{ (rounded to 57)}$$

$$TP = 114 = 72H$$

To change the baud rate of the software UART, use the value of the TP obtained in the following POKE state:

POKE(@90A9,#72)

The maximum baud rate is 1200. The addresses of the UART or bit banger are one-byte changes to make the POKE statements simple. The least significant bit of the echo flag/baud rate is set for 0 for full duplex operation and to 1 for half duplex.

The maximum baud rate is 1200. The addresses of the UART or bit banger are one-byte changes to make the POKE statements simple. Bit 8 of the echo flag/baud rate is set for 0 for full duplex operation and to 1 for half duplex.

Memory Requirements

The interpreter can reside in any 12-kilobyte block of memory provided the user ROM and RAM can be placed above it. The user program resides in ROM immediately following the Interpreter. RAM must be above the user program.

Provisions have been made for the installation of an RCA COSMAC Microboard Video-Audio-Keyboard Interface CDP18S661B at memory address F400H - FFFFH. A typical configuration is given in Fig. 5.

```
10 POKE (@908B,#2):REM CHANGE INPUT TO UART
20 POKE (@908F,#02):REM CHANGE OUTPUT TO UART
30 POKE (@90A9,#00):REM CHANGE ECHO FLAG/BAUD TO UART
```

Fig. 4 - Example showing use of POKE statements.

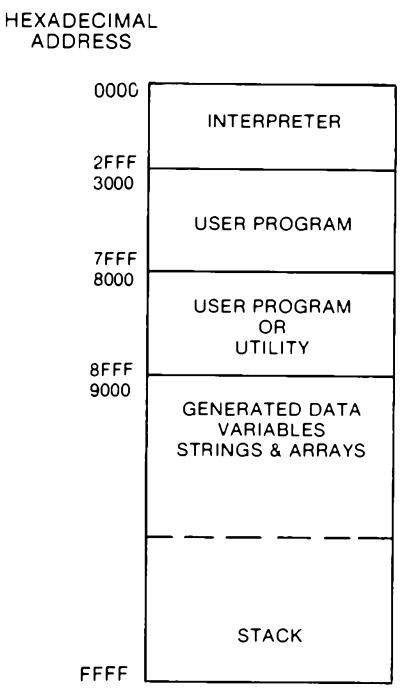


Fig. 5(a) - Typical memory configuration for Run-time BASIC.

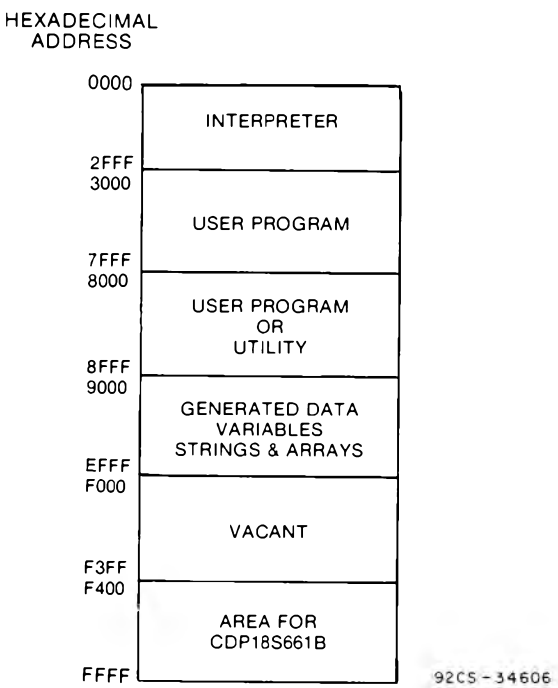


Fig. 5(b) - Typical memory configuration for Run-time BASIC with CDP18S661B interface.

Summary of Run-Time BASIC Usage

Following is a summary of the steps the user should take in putting a BASIC3 program into an EPROM to work with Run-time BASIC.

1. With BASIC3, develop and debug a program on MCDS.
2. Load the BASIC3 program into the Development System and run off a listing to make sure it is correct.
3. Type EOP to find the address of the end of the program. Record that address.
4. Run the program.
5. Type EOD to find the address of the end of the data. Record that address.
6. Subtract 0200H (hexadecimal) from the recorded EOP address (Remember that the subtraction is hexadecimal). The number remaining is the length of program.
7. Subtract the EOP address from the EOD address (both hexadecimal). The value obtained is the RAM space required for the data. The RAM must start at 9000H. BASIC3 searches upward for available RAM starting at 9000H and establishes the stack at the top of this contiguous area. The two work pages start at 9000H and the data starts at 9200H. The minimum RAM area, therefore, is the space required for the data plus enought room for work pages and stack. In general, one page (256 bytes) is enough stack. If nested loops are used in the stack, however, additional room will be needed.
8. Exit from BASIC3 by means of the BYE command.
9. Use the monitor tape write command W to copy onto tape memory from 0200 through the EOP address found in step 3 above.
10. Read in the PROM Programmer tape, verify an empty EPROM, and enter the Program (P) mode. When the query "LOWEST PROM ADDRESS (XX00)" is received for the first EPROM, enter the address 0200H. If additional EPROM's are needed to hold the program, enter the appropriate addresses for them. With 2-kilobyte EPROM's, for example, the second one starts at 0A00H.
11. The EPROM's should now be ready. Remember that the first EPROM must be located at address 3000H in the final system.

Appendix A—
Error Messages and Their Meanings

00 Break message
01 Syntax error in ASC or LEN function
02 Array out of range or not dimensioned at all
03 DIM error; either syntax error or memory overflow
04 DEFINT has illegal line end
05 Missing parentheses in function argument
06 Argument out of range
07 Mixed-mode calculation not permitted
08 Divide by zero error or log of a negative number
09 Non-executable function encountered
10 Cannot use EXIT command unless in GOSUB or FOR/NEXT
11 Stack overflow in FOR/NEXT, or FOR executed directly (unacceptable to execute FOR directly)
12 Syntax error in FOR
13 Stack overflow in GOSUB
14 Unacceptable character in hexadecimal field
15 Floating-point number too large to be converted to integer, or integer multiply overflow
16 Unacceptable operator in conditional statement
17 Can not execute INPUT statement or FVAL in direct mode
18 Must have variable or string name in READ
19 Syntax error in READ or INPUT
20 Syntax error in LEN function
21 Syntax error in LET statement
22 Missing quote
23 Syntax error in LIST
24 No such word found in BASIC3 library
25 Syntax error in MID$ function
26 Unacceptable variable name found in NEXT statement
27 Either a number or a letter was expected
28 Missing arithmetic parentheses
29 Wrong number of arguments in POKE statement
30 Unacceptable last character in PRINT statement
31 Syntax error in DATA statement
32 No more data found
33 No such string found in INPUT statement
34 Missing equal sign in LET with strings
35 Missing parentheses in string array entry
36 Too many arguments in USR or CALL
37 Syntax error in CHR$ function
38 Unacceptable character in binary field (1 or 0)
39 Intermediate line buffer overflow
40 Cannot set tape as input device (file not opened)
41 Cannot set tape as output device (file not opened)
42 Unacceptable line end or non-executable statement
43 Stack overflow in trying to call interrupt routine
44 Number entered has too many digits (10)
45 Unacceptable character in number field
46 No such line number found
47 Unacceptable operator in IF involving strings
48 Memory overflow in trying to load string
49 Wrong number of arguments in MOD function
50 File entered is too large for memory
51 Argument value out of range for INP or OUT
52 Wrong number of arguments in INP function
53 Wrong number of arguments in OUT statement
54 No such string variable yet defined
55 Unable to open READ tape or tape READ error
56 Unable to open WRITE tape or tape WRITE error
57 Unacceptable file type (not a BASIC3 program)
58 Unacceptable file type (not BASIC3 data)
59 Not used
60 Not used
61 Not used
62 Not used
63 Not enough memory for RENUMBER table
64 No such line number found for RENUMBER (Line # may now be erroneous)
65 No such subscripted variable yet defined.
66 Attempting to build a string over 127 characters long

Appendix B—
Backus-Naur Format (BNF)

BNF notation is a concise and convenient way to express the syntax of a language. There are two major elements in notation: terminal and non-terminal elements. A terminal element is written exactly as it would appear when used; a non-terminal element is a description of something and always appears between <>.

Example:

<FIRST THREE LETTERS OF THE ALPHABET> ::= ABC

ABC is not a description of the item, it is the item itself. There are no commas between the letters, because a comma is not part of the alphabet. Likewise, there are no spaces between the letters as spaces are not part of the alphabet. "First three letters of the alphabet" is a description and appears between angle brackets, <>. The symbol ::= can be read as "is defined as" and will be used in every definition. Where there is a choice between alternatives, the symbol | will be used to separate the choice. The square brackets [] denote optional.

Examples:

<one> ::= 1
<plus sign> ::= +
<minus sign> ::= −
<tree> ::= <woody plant>
<binary digit> ::= 0|1

A binary digit could be either a 0 or a 1, but not both. A binary digit can be only a 0 or a 1. A decimal digit can be defined in two ways.

<decimal digit> ::= 0|1|2|3|4| 5|6|7|8|9
<decimal digit> ::= <binary digit>|2|3 |4|5|6|7|8|9

Notice that the decimal digit could be defined by explicitly listing every possibility or by defining it in terms of already defined objects. The use of the description of a binary digit eliminates the need to explicitly list 0 and 1.

Examples:

<primary color> ::= <red>|<green>|<blue>
<American coin names> ::= PENNY|NICKEL
|DIME|HALF-DOLLAR

Note that PENNY is the name itself and so is a terminal element. Red as a non-terminal element describes the color, not the name of the color.

<certain breed of dog> ::= <collie>
|<German shepherd>| <beagle>
<certain name of dog> ::= REX|SPOT
|SHAD|ROVER

If it were necessary to list every possible combination explicitly, BNF would be an extremely voluminous description of anything. Fortunately, it is possible to describe an item recursively, using its own description as part of the description. An unsigned binary number can be defined recursively as follows:

<unsigned binary number> ::= <binary digit>
| <binary digit> <unsigned binary number>

Under this definition 01 is an unsigned binary number because it is 0 (a binary digit) followed by 1 (an unsigned binary number) and 1 is an unsigned binary number because it is 1 (a binary digit). Both the first and second part of the definition were used. 03 is not an unsigned binary number because 03 is not a binary digit (0 or 1 only), and though 0 is a binary digit it is not followed by an unsigned binary number. Because 03 does not satisfy either of the alternatives of the definition, it is not an unsigned binary number. Notice that under the definition of an unsigned binary number, any string of 1''s and 0's of any length is an unsigned binary number. In practice the computer has finite capacity and there are usually additional restrictions. These restrictions will be given as notes in the text.

Examples:

<forest> ::= <tree>|<tree><forest>
<crowd> ::= <person>|<person><crowd>

In reading BNF notation blanks are ignored. Where a blank is required by the syntax of the language, the special character is used. In order to improve the readability of the BNF in the text, many of the spaces have been removed. If there is a question concerning syntax, the syntax description in this Appendix is complete and should be referred to.

It is important to remember that the assembler will be interpreting the program instruction using the syntax described in this manual.

Basic Definitions

Character Set

BASIC3's character set includes all ASCII characters from 00H to 7FH (SOH to DEL), although commands are only recognizable in upper case.

Character String

A character is any of the characters in BASIC3's character set. A character string is any sequence of characters. Any valid line of BASIC3 language is a character string.

<character> ::=Any printing ASCII character and the space character
<character string> ::=<character>
|<character string><character>

Examples:

DFJSHRJQGQH	Character string
F	Character and character string
1	Character and character string

Constants

BASIC3 recognizes two types of constants: numeric and literal. A literal constant is simply any character string between quotes. The exception is that a quote may not appear in a literal constant because this special character would prematurely end the constant. A common error is to forget the closing quote on a literal constant. BASIC3 then considers the line to be an error.

<literal constant>::="<character string>"

A literal constant can be no more than 95 characters long. It may not contain a double quote and is ASCII encoded.

There are four types of numeric constants: binary, decimal, 2-character hexadecimal, and 4-character hexadecimal. A binary constant is an 8-character string of 1's and 0's enclosed in slant rules . A decimal constant is a string of decimal digits (0 - 9). A 2-character hexadecimal constant is a "#" immediately followed by hexadecimal digits (0 - 9, A, B, C, D, E, and F). A four-character hexadecimal constant is a "@" immediately followed by four hexadecimal characters. BASIC3 assumes that a numeric constant is a decimal constant unless it has a special character. All numeric constants are stored as floating-point unless otherwise specified.

<binary digit>::=0|1
<decimal digit>::=binary digit>|2|3
|4|5|6|7|8|9
<hexadecimal digit>::=<decimal digit>
|A|B|C|D|E|F
<decimal constant> ::=<decimal digit>
|<decimal digit><decimal constant>
<2-character hexadecimal constant> ::=
<hexadecimal digit> <hexadecimal digit>
<4-character hexadecimal constant> ::=
@<hexadecimal digit> <hexadecimal digit>
<hexadecimal digit> <hexadecimal digit>

Note that spaces are ignored within numeric constants and that spaces within literal constants are considered valid parts of the constants.

Examples:

1	Decimal constant
00000001	Binary constant (must be 8 characters)
#1F	Hexadecimal constant - equivalent to 31 decimal (must be 2 characters)
@00F1	Hexadecimal constant (must be 4 characters)
"ABCD123"	Literal constant

Error:

F1 Interpreted as a variable because it begins with a letter

Lines and Statements

Not all combinations of characters result in valid lines of BASIC3 language just as not all combinations of characters result in valid English sentences. An English sentence is made up of words and, in the same manner, a line of BASIC3 is made up of statements. In the constructs below, △indicates a required space.

<space>::=△|space>△
<statement set>::=<executable line>
|<statement set>:<statement set>
<comment>::=REM<character string>

All lines must end with a carriage return and may or may not be commented.

<line ending>::=<carriage return>
|<comment><carriage return>
<executable line>::=
<line number><statement set><line ending>
|<space><statement set><line ending>

BNF Syntax of BASIC3

The following is a compilation of the full BNF (Backus-Naur Format) description of the COSMAC BASIC3 language.

<binary digit> ::= 0|1
<decimal digit> ::= <binary digit>|2|3
|4|5|6 |7|8|9
<hexadecimal digit> ::= <decimal digit>
|A|B|C |D|E|F
<constant> ::= <binary constant>
|<octal constant>
|<decimal constant>|<hexad ecimal constant>
|<2-character hexadecimal constant>
|<4-character hexadecimal constant>

<relation> ::= <term><relational operator><term>
|<relation><space>|<space><relation>
| NOT <relation>| (<logical term>)

<logical element> ::= <relation>
|<logical element><space>
|<logical element> AND <logical element>
|<space><logical element>

<logical factor> ::= <logical element>
|<logical factor><space>
|<space><logical factor>
|<logical factor> XOR <logical factor>

<logical term> ::= <logical factor>
|<logical term><space>
|<space><logical term>
|<logical term> OR <logical term>

<letter>::=A|B|C |D|. . . . . |Z

<digit>::= 0|1|2| 3|4|5|6 |7|8|9

<sign>::=+|−

<number>::=<integer>|<floating point>

<integer>::=<digit>|<integer> <digit>

<floating point>::=<decimal part>
|<decimal part>[<exponent part>]

<decimal part>::=<integer>[.[<integer>]]
|.<integer>

<exponent part>::=E<integer>
|E<sign><integer>

<delimiter>::=,|;|: |<CR>

<statement number>::=<integer>

<basic program>::=<program statement>
|<basic program><program statement>

<program statement>::=
[<statement number>]<statement list>

<statement list>::=
<statement list>:<basic statement>
|<basic statement>

<basic statement>::=<bye statement>
|<call statement>
|<cld statement>
|<close statement>|<cls statement>
|<data statement>|<defint statement>
|<defus statement>
|<degree statement>
|<dim statement>|<din statement>
|<disint statement>|<dload statement>
|<dmapt statement>
|<dout statement>
|<dsave statement>|<edit statement>
|<end statement>
|<enint statement>
|<eod statement>
|<eop statement>
|<exit statement>|<fixed statement>
|<for statement>|<format statement>
|<gosub statement>
|<goto statement>
|<if statement>|<input statement>
|<let statement>|<list statement>
|<new statement>|<next statement>
|<out statement>|<pload statement>
|<poke statement>
|<print statement>
|<psave statement>
|<radian statement>
|<read statement>|<rem statement>
|<renumber statement>
|<restore statement>
|<return statement>|<rfln statement>
|<run statement>|<stq statement>
|<tin statement>|<tout statement>
|<trace statement>|<wait statement>
|<wfln statement>

<variable>::=<simple variable>
|<subscripted variable>
|<string variable>

<simple variable>::=<letter>

<subscripted variable>::=<letter>(<expression>)

<string variable>::=<letter>$[(<expression>)]

<expression>::=<conjunction>
|<expression>OR<conjunction>

<conjunction>::=<boolean primary>
|<conjunction>AND<boolean primary>

<boolean primary>::=<arithmetic expression>
|<boolean primary> <relational operator>
|<arithmetic expression>

<arithmetic expression>::=<term>
|<arithmetic expression>+<term>
|<arithmetic expression>−<term>

<term>::=<factor>|<term>* <factor>
|<term>/<factor>

<factor>::=<primary>|<sign> <primary>
|<exponentiation>
|NOT(<primary>)

<primary>::=<variable>|<number>

<relational operator>::= = |>|<| <=|> =|<>

<exponentiation>::=<term>↑<term>

<string>::="<character string>"

<parameter part>::=(<expression list>
|<string expression>)

<expression list> ::= <expression list>, <expression>
|<expression>

<variable list>::=<variable>
|<variable list>,<variable>

<function>::=<function name>[<parameter part>]

<function name>::=ABS|ASC| ATN|CHR$|COS
|EF|EXP| FNUM|FVAL|INP| INT|INUM
|LEN|LOG|MEM |MID$|MOD| PEEK|PI
|QST|RND|SGN |SIN|SQR|TAB |USR

<bye statement>::=BYE

<call statement>::= CALL <parameter part>

<cld statement>::=CLD
<close statement>::=CLOSE
<cls statement>::=CLS
<data statement>::=DATA<data list>
<data list>::=<data list>,<data element>
|<data element>
<data element>::=<string>|<expression>
<defint statement>::=DEFINT[<simple variable>]
<defus statement>::=DEFUS<expression>
<degree statement>::=DEG
<dim statement>::=DIM<variable list>
<din statement>::=DIN
<disint statement>::=DISINT
<dload statement>::=DLOAD <tape #>
<dmapt statement>::=DMAPT<expression>
<dout statement>::=DOUT
<dsave statement>::=DSAVE<tape #>
<edit statement>::=EDIT<expression>
<end statement>::=END
<enint statement>::=ENINT<expression>
<eod statement>::=EOD
<eop statement>::=EOP
<exit statement>::=EXIT<expression>
<fixed statement>::=FIXED<expression>
<for statement>::=FOR<simple variable>=
<expression>TO<expression>
[STEP<expression>]
<format statement>::=FORMAT<integer>
<gosub statement>::=GOSUB<expression>
<goto statement>::=GOTO<expression>
<if statement>::=IF<relation>
[THEN]<basic statement>
|IF<string reference>=<string expression>
[THEN]<basic statement>
|IF<string reference>=><string expression>
[THEN]<basic statement>
<string reference>::=<string variable>
|MID$(<string ref>,<expression>
[,<expression>])
<string ref>::=STR$<expression>
|<string variable>+ <string variable>
|<string variable>
|<string variable>+<string>
<input statement>::=INPUT[<string>]
<variable list>
<let statement>::=[LET]<simple variable>
|<subscripted variable>=<expression>
|[LET]<string variable>=
<string expression>
<list statement>::=LIST[<expression>]
[,<expression>]
<new statement>::=NEW
<next statement>::=NEXT[<simple variable>]
<out statement>::=OUT(<expression>
|,<expression>|,<expression>)
<pload statement>::=PLOAD<tape #>
<poke statement>::=POKE(<expression>,
<expression>)
<print statement>::=PRINT[<print list>]
<print list>::=<print list element>
[<print delimiter>]
|<print list><print delimiter>
<print list element>
<print list element>::=CRS$<parameter part>
|TAB(<expression>)
|<expression>
|<string expression>
<print delimiter>::=;|,
<psave statement>::=PSAVE<tape #>
<radian statement>::=RAD
<read statement>::=READ<variable list>
<rem statement>::=REM
<renumber statement>::=RENUMBER[<integer>]
<restore statement>::=RESTORE
<return statement>::=RETURN
<rfln statement>::=RFLN <tape #>
<run statement>::=RUN[<expression>|+]
<stq statement>::=STQ<expression>
<tin statement>::=TIN
<tout statement>::=TOUT
<trace statement>::=TRACE>expression>
<wait statement>::=WAIT<expression>
<wfln statement>::=WFLN[<tape #>]
<string expression>::=<string ref>
|<string>|<string>+<string>

Appendix C—
Work Page Allocation Details

Work Page #1

00-5F	Line buffer
60-7F	Buffer overflow area
80	Backspace character
81-82	Start of user space
83-84	End of user space
85-86	Current line number in line number search routine
87	Input data pointer
88-89	Currently executing line number
8A-8D	Input hook*
8E-91	Output hook*
92-93	End of arrays and start of strings
94-95	Start of arrays
96-97	Line number for interrupt routine
98	Interrupt flag
99-9A	End of string variables
9B	X-coordinate used in TAB function
9C	Degree/radian flag
9D	Tape I/O flags
9E	Fixed start of RAM user space
9F-A2	Statement call
A3-A6	Function call
A7-A8	User as x,y for memory mapped I/O
A9	Echo flag
AA	Delete character
AB	Cancel character
AC	Space character
AD	Clear screen character
AE	EDIT stop character
AF	Flag used in LIST routine
B0	Data available flag
B1-B2	Data pointer
B3	FORMAT information
B4	Reserved
B5	EDIT erase character
B6	EDIT insert character
B7-B8	Reserved
B9	READ**
BA	WRITE**
BB	Stack page
BC	ASCII value of split for fixed/floating point
BD	# of digits in fixed point mode
BE-BF	Random number seed
C0-CB	Number work area
CC	Reserved
CD	CLS delay constant
CE	EDIT change character
CF	EDIT abort character
D0-FF	Intermediate line buffer
F0-F4	Temporary work area for PORT instructions

Work Page #2

00-68	Variable storage
70-FF	Computation stack

*Bytes 8A and 8E are call instructions (D4)
Bytes 8D and 91 are exit instructions (D5)

**Used in setting up read or write buffer

Note:
Work Page #1 starting address is @0000 in BASIC3 and @9000 in Run-time BASIC.

Work Page #2 starting address is @0100 in BASIC3 and @9100 in Run-time BASIC.

Appendix D—
ASCII-Hex Table

MOST SIGNIFICANT HEX DIGIT

LEAST SIGNIFICANT HEX DIGIT	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	—	=	M	]	m	}
E	SO	RS	.	>	N	↑	n	~
F	SI	US	/	?	O	←	o	DEL

NOTES:

(1) Parity bit in most significant hex digit not included.

(2) Characters in columns 0 and 1 (as well as SP and DEL) are non-printing.

(3) Model 33 Teletypewriter prints codes in columns 6 and 7 as if they were column 4 and 5 codes.

92CS-34736

Appendix E—
Summary of Statements

BYE - Return to MCDS
CALL - Call to machine language subroutine
CLD - Clear data
CLOSE - Append DC3 to tape file
CLS - Clear CRT screen
DATA - Program data
DEFINT - Define variables as integers
DEFUS - Relocate start of program space
DEG - Set angle measure to degrees
DIM - Dimensioning of arrays
DIN - Set tape drive as input device
DISINT - Disable interrupts
DLOAD - Read data file from tape
DMAPT - Set up DMA pointer
DOUT - Set tape as output device
DSAVE - Save data in a tape file
EDIT - Edit line
END - Program delimiter
ENINT - Enable interrupts
EOD - End of data
EOP - End of program
EXIT - Loop exit
FIXED - Fraction digits printed
FOR - FOR loop
FORMAT - Set field of numeric data
GOSUB -Subroutine call
GOTO - Unconditional branch
IF - Conditional statement
INPUT - Read from terminal or tape
LET - Assignment statement
LIST - List program to output device
NEW - Clear program space
NEXT - FOR loop termination
OUT - Output data to a port
PLOAD - Read program file from tape
POKE - Direct memory addressing
PRINT or PR - Output data to terminal or tape
PSAVE - Save program in a tape file
RAD - Set angle measure to radians
READ - Read program data
REM - Comment lines
RENUMBER - Renumber program lines
RESTORE - Reset program data pointer
RETURN - Return from subroutine
RFLN - Open read file
RUN - Run program
RUN+ - Run program in fast mode
STQ - Set Q
TIN - Set terminal as input device
TOUT - Set terminal as output device
TRACE - List line numbers as executed
WAIT - Delay
WFLN - Open write file

Appendix F—
Summary of Functions

ABS - absolute
ASC - ASCII value
ATN - arctangent
CHR$ - ASCII output
COS - cosine
EF - value of flag EF1 through EF4
EXP - e
FNUM - convert number to floating point
FVAL - evaluate string as expression
INP - read from input port
INT - return integer portion of floating-point number
INUM - convert number to integer
LEN - string length
LOG - ln x
MEM - calculate free RAM
MID$ - string pointer
MOD - modulo
PEEK - direct memory addressing
PI - 3.14159
QST - value of Q line
RND - random number
SGN - sign value
SIN - sine
SQR - square root
TAB - printing format control
USR - machine language functions

Appendix G— Alphabetical Index of BASIC3 Statements and Functions

ABS	23	GOSUB	18	RAD	17
ASC	24	GOTO	18	READ	20
ATN	23			REM	17
		IF	19	RENUMBER	15
BYE	13	INP	25	RESTORE	20
		INPUT	20	RETURN	19
CALL	22	INT	23	RFLN	22
CHR$	24	INUM	23	RND	24
CLD	13			RUN	15
CLOSE	22	LEN	24	RUN+	15
CLS	13	LET	17		
COS	23	LIST	15	SGN	24
		LOG	23	SIN	24
DATA	19			SQR	24
DEFINT	16	MEM	25	STQ	17
DEFUS	16	MID$	25		
DEG	16	MOD	23	TAB	25
DIM	16			TIN	22
DIN	22	NEW	15	TOUT	22
DISINT	14	NEXT	29	TRACE	15
DLOAD	21				
DMAPT	17	OUT	20	USR	26
DOUT	22				
DSAVE	21	PEEK	25	WAIT	19
EDIT	13	PI	24	WFLN	22
EF	25	PLOAD	21		
END	17	POKE	17		
ENINT	14	PR	21		
EOD	14	PRINT	21		
EOP	14	PSAVE	21		
EXIT	17				
EXP	23	QST	25		
FIXED	17				
FNUM	23				
FOR	18				
FORMAT	15				
FVAL	24				

Appendix H—
Using the Cassette Tapes

The cassette tape drives (0 and 1) are under software control through the REMOTE jack. When the user plans to read or write a tape for the first time, he should first make sure that the cassettes are rewound. The UT62 monitor program provides for rewinding through the J command. This command enables the REMOTE jacks of tape 0 and tape 1 so that the control buttons (play, rewind, fast-forward, etc.) are active. When tape control is restored to the system, the user should press the PLAY (READ) or RECORD (WRITE) buttons prior to issuing commands to the tape.

The J command is available in the UT62 monitor, the Editor, and the Assembler but not in BASIC3. To enable the tapes in BASIC3, the user can call the J routine by typing:

```
CALL(@8504)
```

After the tape is rewound, the user can press any key to return to BASIC3.

Reference should also be made to the **User Manual for the RCA COSMAC Microboard Computer Development Systems (MCDS) CDP18S693 and CDP18S694,** MPM-293, for additional information on the cassette-tape interface.

Appendix I—
Program for Converting from Pseudocode into ASCII.

```
!M
0000 ;              0001 ..PROGRAM TO CONVERT PSEUDOCODE TO ASCII FOR
0000 ;              0002 ..
0000 ;              0003 ..RUNTIME BASIC EPROM/ROM APPLICATION.
0000 ;              0004
0000 ;              0005         ..REGISTER DEFINITIONS
0000 ;              0006 T0=0                              ..TEMPORARY REGISTER
0000 ;              0007 INT=1                             ..INTERRUPT POINTER
0000 ;              0008 SP=2                              ..STACK POINTER
0000 ;              0009 PC=3                              ..PROGRAM POINTER
0000 ;              0010 CR=4                              ..CALL REGISTER
0000 ;              0011 RR=5                              ..RETURN REGISTER
0000 ;              0012 ARG=6                             ..ARGUMENT POINTER
0000 ;              0013 T1=7                              ..TEMPORARY REGISTER
0000 ;              0014 T2=8                              ..TEMPORARY REGISTER
0000 ;              0015 G0=9                              ..GLOBAL REGISTER
0000 ;              0016 G1=10                             ..GLOBAL REGISTER
0000 ;              0017 ADDRSS=#0B                        ..COUNTER FOR ADDRESS
0000 ;              0018 COUNT=8                           ..COUNT FOR NO. BYTES/LINE
0000 ;              0019 CHAR=#0F
0000 ;              0020 CNT=#0D                           ..COUNT FOR NO. OF BYTES IN FILE
0000 ;              0021                 ..TAPE UTILITY EQUATES
0000 ;              0022 TREAD=#8502                       ..RETURN BYTE FROM TAPE IN RF.1
0000 ;              0023 TWRITE=#8500                      ..PASS BYTE IN RF.1 TO TAPE
0000 ;              0024 REWIND=#8504                      ..ADDRESS OF TAPE REWIND IN UTILITY
0000 ;              0025
0000 ;              0026                 ..UTILITY ADDRESSES
0000 ;              0027 INIT1=#83F3                       ..INIT SCRT, GOTO #0005
0000 ;              0028 READ=#813E                        ..CHAR->RF.1
0000 ;              0029 READAH=#813B                      ..CHAR->RF.1, IF HEX DIGIT->RD
0000 ;              0030 TYPE5=#81A0                       ..M(R5)->TTY
0000 ;              0031 TYPE6=#81A2                       ..M(R6)->TTY
0000 ;              0032 TYPE=#81A4                        ..RF.1->TTY
0000 ;              0033 TYPE2=#81AE                       ..HEX-PAIR(RF.1)->TTY
0000 ;              0034 OSTRNG=#83F0                      ..STRING AT M(R6) -> TTY
0000 ;              0035 CKHEX=#83FC                       ..SEE IF RF.1 IS HEX DIGIT
0000 ;              0036 GOUTIL=#83F9                      ..RETURN TO UT62
0000 ;              0037
0000 ;              0038                 ..START OF PROGRAM
0000 ;              0039
0000 7100;          0040         DIS;,0                    ..INTERRUPTS OFF
0002 C083F3;        0041         LBR INIT1                 ..INITIALIZE CALL+RETN
0005 D483F0;        0042         SEP R4,A(OSTRNG)
0008 0D0A504C414345;0043         ,#0D0A,T'PLACE SOURCE FILE IN TAPE DRIVE 0'
000F 20534F55524345;0043
0016 2046494C452049;0043
001D 4E205441504520;0043
0024 44524956452030;0043
002B 0D0A504C414345;0044         ,#0D0A,T'PLACE OUTPUT FILE IN TAPE DRIVE 1'
0032 204F5554505554;0044
0039 2046494C452049;0044
```

```
0040 4E205441504520;0044
0047 44524956452031;0044
004E 0D0A00;        0045        ,#0D0A,0
0051 D48504;        0046        SEP R4,A(REWIND)        ..REWIND TAPES
0054 ;              0047
0054 ;              0048                                ..MAIN WRITE LOOP
0054 F830BB;        0049        LDI #30;PHI ADDRSS
0057 F800ABA8;      0050        LDI 0;PLO ADDRSS;PLO COUNT
005B D400F0;        0051        SEP R4,A(GETTER)
005E FB11;          0052        XRI #11                 ..TEST FOR PSAVE OR DSAVE
0060 CA0138;        0053        LBNZ EOF                ..DC1 PSAVE
0063 F821BF;        0054        LDI T'!';PHI CHAR
0066 D40122;        0055        SEP R4,A(PUTTER)
0069 F84DBF;        0056        LDI T'M';PHI CHAR
006C D40122;        0057        SEP R4,A(PUTTER)
006F F80DBF;        0058        LDI#0D;PHI CHAR
0072 D40122;        0059        SEP R4,A(PUTTER)
0075 F80ABF;        0060        LDI #0A;PHI CHAR
0078 D40122;        0061        SEP R4,A(PUTTER)
007B 9BBF;          0062        GHI ADDRSS PHI CHAR
007D D400C9;        0063        SEP R4,A(DISMBL)
0080 8BBF;          0064        GLO ADDRSS PHI CHAR
0082 D400C9;        0065        SEP R4,A(DISMBL)
0085 F820BF;        0066        LDI T' ' PHI CHAR
0088 D40122;        0067        SEP R4,A(PUTTER)
008B D400F0;        0068        SEP R4,A(GETTER)
008E BD;            0069        PHI CNT                 ..HIGH HALF OF NO. OF BYTES ON FILE
008F D400F0;        0070        SEP R4,A(GETTER)
0092 AD;            0071        PLO CNT                 ..LOW HALF OF NO. OF BYTES IN FILE
0093 D400F0;        0072 DOIT:  SEP R4,A(GETTER)
0096 D400C9;        0073        SEP R4,A(DISMBL)
0099 1B182D;        0074        INC ADDRSS;INC COUNT;DEC CNT
009C 88FB20;        0075        GLO COUNT;XRI 32        ..32 HEX BYTES PER LINE
009F 3A93;          0076        BNZ DOIT
00A1 F83BBF;        0077        LDI T';';PHI CHAR
00A4 D40122;        0078        SEP R4,A(PUTTER)
00A7 F80DBF;        0079        LDI #0D;PHI CHAR
00AA D40122;        0080        SEP R4,A(PUTTER)
00AD F80ABF;        0081        LDI #0A;PHI CHAR
00B0 D40122;        0082        SEP R4,A(PUTTER)
00B3 9BBF;          0083        GHI ADDRSS;PHI CHAR
00B5 D400C9;        0084        SEP R4,A(DISMBL)
00B8 8BBF;          0085        GLO ADDRSS PHI CHAR
00BA D400C9;        0086        SEP R4,A(DISMBL)
00BD F820BF;        0087        LDI T' ';PHI CHAR
00C0 D40122;        0088        SEP R4,A(PUTTER)
00C3 F800A8;        0089        LDI 0;PLO COUNT
00C6 C00093;        0090        LBR DOIT
00C9 ;              0091
00C9 9F73;          0092 DISMBL: GHI CHAR;STXD
00CB F6F6F6F6;      0093        SHR;SHR;SHR;SHR
00CF BF;            0094        PHI CHAR
00D0 D400E2;        0095        SEP R4,A(CHKHEX)
00D3 D40122;        0096        SEP R4,A(PUTTER)
00D6 1202;          0097        INC SP;LDN SP
00D8 FA0F;          0098        ANI #0F
00DA BF;            0099        PHI CHAR
00DB D400E2;        0100        SEP R4,A(CHKHEX)
00DE D40122;        0101        SEP R4,A(PUTTER)
00E1 D5;            0102        SEP R5
00E2 ;              0103
00E2 9FFF0A;        0104 CHKHEX: GHI CHAR;SMI 10
00E5 3BEB;          0105        BNF CHKHX1
00E7 FC41BF;        0106        ADI #41;PHI CHAR
00EA D5;            0107        SEP R5
00EB 9FF930BF;      0108 CHKHX1: GHI CHAR;ORI #30;PHI CHAR
00EF D5;            0109        SEP R5
```

```
00F0 ;                  0110
00F0 F84EAC;            0111 GETTER: LDI A.0(IOCB1);PLO RC
00F3 D48502;            0112         SEP R4,A(TREAD)
00F6 9D;                0113         GHI CNT
00F7 CA00FE;            0114         LBNZ GET1
00FA 8D;                0115         GLO CNT
00FB C20101;            0116         LBZ FIX1                ..STACK BLOWN
00FE 80B0;              0117 GET1:   GLO T0;PHI T0           ..SAVE THIS STATUS
0100 D5;                0118 GETDON: SEP R5
0101 ;                  0119
0101 F80DBF;            0120 FIX1:   LDI #0D;PHI CHAR
0104 D40122;            0121         SEP R4,A(PUTTER)
0107 F80ABF;            0122         LDI #0A;PHI CHAR
010A D40122;            0123         SEP R4,A(PUTTER)
010D F800BF;            0124         LDI 0;PHI CHAR
0110 D400C9;            0125         SEP R4,A(DISMBL)        ..'0000' AT END OF FILE FOR FORMAT
0113 F800BF;            0126         LDI 0;PHI CHAR
0116 D400C9;            0127         SEP R4,A(DISMBL)
0119 F813BF;            0128         LDI #13;PHI CHAR        ..DC3
011C D40122;            0129         SEP R4,A(PUTTER)
011F C00138;            0130         LBR EOF
0122 ;                  0131
0122 F851AC;            0132 PUTTER: LDI A.0(IOCB2);PLO RC
0125 D48500;            0133         SEP R4,A(TWRITE)
0128 C30138;            0134         LBDF EOF
012B 9FFB13C20138;      0135         GHI CHAR;XRI #13;LBZ EOF
0131 90FB33C20138;      0136         GHI T0;XRI #33;LBZ EOF
0137 D5;                0137         SEP R5
0138 ;                  0138
0138 ;                  0139                 ..CLOSE FILES
0138 F851AC;            0140 EOF:    LDI A.0(IOCB2);PLO RC
013B 1C;                0141         INC RC
013C F881;              0142         LDI #81
013E 5C;                0143         STR RC
013F 1C;                0144         INC RC
0140 F87F;              0145         LDI #7F
0142 5C;                0146         STR RC
0143 F813BF;            0147         LDI #13;PHI CHAR
0146 2C2C;              0148         DEC RC;DEC RC
0148 D48500;            0149         SEP R4,A(TWRITE)
014B ;                  0150                 ..END OF PROGRAM
014B C083F9;            0151 EOP:    LBR GOUTIL
014E ;                  0152
014E ;                  0153         ..IOCB'S
014E 00;                0154 IOCB1:  ,0                      ..BLOCK #
014F 40;                0155         ,#40                    ..TAPE DRIVE #
0150 00;                0156         ,0                      ..COUNT
0151 00;                0157 IOCB2:  ,0                      ..BLOCK #
0152 80;                0158         ,#80                    ..TAPE DRIVE #
0153 00;                0159         ,0                      ..COUNT
0154 ;                  0160         END
0000

>
```

An alphabetical index of BASIC3 Statements and Functions is given in Appendix G on page 42.

A listing of the error messages and their meanings is given in Appendix A on page 33.